

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_ARRAY_H
22: #define SPL_ARRAY_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26: #include "spl_iterators.h"
27:
28: extern ZEND_API zend_class_entry *spl_ce_ArrayObject;
29: extern ZEND_API zend_class_entry *spl_ce_ArrayIterator;
30: extern ZEND_API zend_class_entry *spl_ce_RecursiveArrayIterator;
31:
32: PHP_MINIT_FUNCTION(spl_array);
33:
34: extern void spl_array_iterator_append(zval *object, zval *append_value);
35: extern void spl_array_iterator_key(zval *object, zval *return_value);
36:
37: #endif /* SPL_ARRAY_H */
38:
39: /*
40:  * Local Variables:
41:  * c-basic-offset: 4
42:  * tab-width: 4
43:  * End:
44:  * vim600: fdm=marker
45:  * vim: noet sw=4 ts=4
46:  */
```

```
1: /*
2:  *
3:  * PHP Version 7
4:  *
5:  * Copyright (c) 1997-2018 The PHP Group
6:  *
7:  * This source file is subject to version 3.01 of the PHP license,
8:  * that is bundled with this package in the file LICENSE, and is
9:  * available through the world-wide-web at the following url:
10:  * http://www.php.net/license/3.01.txt
11:  * If you did not receive a copy of the PHP license and are unable to
12:  * obtain it through the world-wide-web, please send a note to
13:  * license@php.net so we can mail you a copy immediately.
14:  *
15:  * Authors: Etienne Kneuss <colder@php.net>
16:  */
17:
18:
19: /* $Id$ */
20:
21: #ifdef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "php.h"
26: #include "zend_exceptions.h"
27:
28: #include "php_spl.h"
29: #include "spl_functions.h"
30: #include "spl_engine.h"
31: #include "spl_iterators.h"
32: #include "spl_heap.h"
33: #include "spl_exceptions.h"
34:
35: #define PTR_HEAP_BLOCK_SIZE 64
36:
37: #define SPL_HEAP_CORRUPTED 0x00000001
38:
39: #define SPL_PRIORITY_EXTR_MARK 0x00000003
40: #define SPL_PRIORITY_EXTR_BOTH 0x00000003
41: #define SPL_PRIORITY_EXTR_DATA 0x00000001
42: #define SPL_PRIORITY_EXTR_PRIORITY 0x00000002
43:
44: zend_object_handlers spl_handler_SplHeap;
45: zend_object_handlers spl_handler_SplPriorityQueue;
46:
47: PHPAPI zend_class_entry *spl_ow_SplHeap;
48: PHPAPI zend_class_entry *spl_ow_SplMaxHeap;
49: PHPAPI zend_class_entry *spl_ow_SplMinHeap;
50: PHPAPI zend_class_entry *spl_ow_SplPriorityQueue;
51:
52:
53: typedef void (*spl_ptr_heap_dtor_func)(zval *);
54: typedef void (*spl_ptr_heap_ctor_func)(zval *);
55: typedef int (*spl_ptr_heap_cmp_func)(zval *, zval *, zval *);
56:
57: typedef struct _spl_ptr_heap {
58:     zval *elements;
59:     spl_ptr_heap_ctor_func ctor;
60:     spl_ptr_heap_dtor_func dtor;
61:     spl_ptr_heap_cmp_func cmp;
62:     int count;
63:     int max_size;
64:     int flags;
65: } spl_ptr_heap;
66:
67: typedef struct _spl_heap_object spl_heap_object;
68: typedef struct _spl_heap_it spl_heap_it;
69:
70: struct _spl_heap_object {
71:     spl_ptr_heap heap;
72:     int flags;
73:     zend_class_entry *ce;
74:     zend_function *ctor;
75:     zend_function *dtor;
76:     zend_object *obj;
77: };
78:
79: /* define an overloaded iterator structure */
80: struct _spl_heap_it {
81:     zend_user_iterator intern;
82:     int flags;
83: };
84:
85: static inline spl_heap_object *spl_heap_from_obj(zend_object *obj) /* {{{ */ {
86:     return (spl_heap_object *) (char *) (obj - XOffsetof(spl_heap_object, std));
87: } /* }}} */
88:
89:
90: #define Z_SPLHEAP_P(rv) spl_heap_from_obj(Z_OBJ_P(rv))
91:
92: static void spl_ptr_heap_dtor(zval *elem) /* {{{ */ {
93:     if (!IS_UNDEF_P(elem)) {
94:         spl_ptr_heap_dtor(elem);
95:     }
96: } /* }}} */
97:
98:
99: static void spl_ptr_heap_ctor(zval *elem) /* {{{ */ {
100:     Z_TRY_ADDREF_P(elem);
101: } /* }}} */
102:
103:
104: static int spl_ptr_heap_cmp_cb_helper(zval *obj, spl_heap_object *heap, zval *a, zval *b, zend_long *result) /* {{{ */ {
105:     zval result;
106:
107:     zend_call_method_with_2_params(obj, heap->std.ce, &heap->std.ce, "compare", &result, a, b);
108:
109:     if (EG(exception)) {
110:         return FAILURE;
111:     }
112:
113:     *result = zend_get_long(&result);
114:     spl_ptr_heap_dtor(&result);
115:
116:     return SUCCESS;
117: } /* }}} */
118:
119:
120: static zval *spl_pqueue_extract_helper(zval *value, int flags) /* {{{ */ {
121:     if ((flags & SPL_PRIORITY_EXTR_BOTH) == SPL_PRIORITY_EXTR_BOTH) {
122:         return value;
123:     } else if ((flags & SPL_PRIORITY_EXTR_BOTH) > 0) {
124:         if ((flags & SPL_PRIORITY_EXTR_DATA) == SPL_PRIORITY_EXTR_DATA) {
125:             zval *data;
126:             if ((data = zend_hash_str_find(Z_ARRVAL_P(value), "data", sizeof("data") - 1)) != NULL) {
127:                 return data;
128:             }
129:         } else {
130:             zval *priority;
131:             if ((priority = zend_hash_str_find(Z_ARRVAL_P(value), "priority", sizeof("priority") - 1)) != NULL) {
132:                 return priority;
133:             }
134:         }
135:     }
136:
137:     return NULL;
138: } /* }}} */
139:
140:
141: static int spl_ptr_heap_zval_max_cmp(zval *a, zval *b, zval *obj) /* {{{ */ {
142:     zval result;
143:
144:     if (EG(exception)) {
145:         return 0;
146:     }
147:
148:     if (obj) {
149:         spl_heap_object *heap_obj = Z_SPLHEAP_P(obj);
150:         if (heap_obj->std.ce == SPL_HEAP_P(obj)) {
151:             zend_long lval = 0;
152:             if (spl_ptr_heap_cmp_cb_helper(obj, heap_obj, a, b, &lval) == FAILURE) {
153:                 /* exception or call failure */
154:                 return 0;
155:             }
156:         }
157:         return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
158:     }
159:
160:     compare_function(&result, a, b);
161:     return (int) Z_LVAL(result);
162: } /* }}} */
163:
164:
165: static int spl_ptr_heap_zval_min_cmp(zval *a, zval *b, zval *obj) /* {{{ */ {
166:     zval result;
167:
168:     if (EG(exception)) {
169:         return 0;
170:     }
171:
172:     if (obj) {
173:         spl_heap_object *heap_obj = Z_SPLHEAP_P(obj);
174:         if (heap_obj->std.ce == SPL_HEAP_P(obj)) {
175:             zend_long lval = 0;
176:             if (spl_ptr_heap_cmp_cb_helper(obj, heap_obj, a, b, &lval) == FAILURE) {
177:                 /* exception or call failure */
178:                 return 0;
179:             }
180:         }
181:         return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
182:     }
183:
184:     compare_function(&result, b, a);
185:     return (int) Z_LVAL(result);
186: } /* }}} */

```

```
189:
190: static int spl_ptr_pqueue_zval_cmp(zval *a, zval *b, zval *obj) /* {{{ */ {
191:     zval result;
192:     zval *a_priority_p = spl_pqueue_extract_helper(a, SPL_PRIORITY_EXTR_PRIORITY);
193:     zval *b_priority_p = spl_pqueue_extract_helper(b, SPL_PRIORITY_EXTR_PRIORITY);
194:
195:     if (!IS_NULL(a_priority_p) || !IS_NULL(b_priority_p)) {
196:         zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
197:         return 0;
198:     }
199:
200:     if (EG(exception)) {
201:         return 0;
202:     }
203:
204:     if (obj) {
205:         spl_heap_object *heap_obj = Z_SPLHEAP_P(obj);
206:         if (heap_obj->std.ce == SPL_HEAP_P(obj)) {
207:             zend_long lval = 0;
208:             if (spl_ptr_heap_cmp_cb_helper(obj, heap_obj, a, b, &lval) == FAILURE) {
209:                 /* exception or call failure */
210:                 return 0;
211:             }
212:             return lval > 0 ? 1 : (lval < 0 ? -1 : 0);
213:         }
214:     }
215:
216:     compare_function(&result, a, b);
217:     return (int) Z_LVAL(result);
218: } /* }}} */
219:
220:
221: static spl_ptr_heap *spl_ptr_heap_init(spl_ptr_heap_cmp_func cmp, spl_ptr_heap_ctor_func ctor, spl_ptr_heap_dtor_func dtor) /* {{{ */ {
222:     spl_ptr_heap *heap = emalloc(sizeof(spl_ptr_heap));
223:
224:     heap->dtor = dtor;
225:     heap->ctor = ctor;
226:     heap->cmp = cmp;
227:     heap->elements = malloc(PTR_HEAP_BLOCK_SIZE, sizeof(zval));
228:     heap->max_size = PTR_HEAP_BLOCK_SIZE;
229:     heap->count = 0;
230:     heap->flags = 0;
231:
232:     return heap;
233: } /* }}} */
234:
235:
236: static void spl_ptr_heap_insert(spl_ptr_heap *heap, zval *elem, void *comp_userdata) /* {{{ */ {
237:     int i;
238:
239:     if (heap->count > heap->max_size) {
240:         /* we need to allocate more memory */
241:         heap->elements = realloc(heap->elements, heap->max_size * 2 * sizeof(zval));
242:         memset(heap->elements + heap->max_size, 0, heap->max_size * sizeof(zval));
243:         heap->max_size *= 2;
244:     }
245:
246:     /* sifting up */
247:     for (i = heap->count; i > 0 & heap->cmp(heap->elements[(i-1)/2], elem, comp_userdata) < 0; i = (i-1)/2) {
248:         heap->elements[i] = heap->elements[(i-1)/2];
249:     }
250:     heap->count++;
251:
252:     if (EG(exception)) {
253:         /* exception thrown during comparison */
254:         heap->flags |= SPL_HEAP_CORRUPTED;
255:     }
256:
257:     ZVAL_COPY_VALUE(heap->elements[i], elem);
258:
259:     /* sifting up */
260: } /* }}} */
261:
262: static zval *spl_ptr_heap_top(spl_ptr_heap *heap) /* {{{ */ {
263:     if (heap->count == 0) {
264:         return NULL;
265:     }
266:
267:     return Z_UNDEF(heap->elements[0]) ? NULL : heap->elements[0];
268: } /* }}} */
269:
270:
271: static void spl_ptr_heap_delete_top(spl_ptr_heap *heap, zval *elem, void *comp_userdata) /* {{{ */ {
272:     int i, j;
273:     zend_long limit = (heap->count-1)/2;
274:     zval *bottom;
275:
276:     if (heap->count == 0) {
277:         ZVAL_UNDEF(elem);
278:         return;
279:     }
280:
281:     ZVAL_COPY_VALUE(elem, heap->elements[0]);
282:     bottom = heap->elements[--heap->count];
283:
284:     for (i = 0; i < limit; i++) {
285:         /* find smaller child */
286:         j = i * 2 + 1;
287:         if (j > heap->count & heap->cmp(heap->elements[j], heap->elements[i], comp_userdata) > 0) {
288:             /* next child is bigger */
289:             break;
290:         }
291:         /* swap elements between two levels */
292:         if (heap->cmp(bottom, heap->elements[j], comp_userdata) < 0) {
293:             heap->elements[i] = heap->elements[j];
294:         } else {
295:             break;
296:         }
297:     }
298:
299:     if (EG(exception)) {
300:         /* exception thrown during comparison */
301:         heap->flags |= SPL_HEAP_CORRUPTED;
302:     }
303:
304:     ZVAL_COPY_VALUE(heap->elements[i], bottom);
305:
306:     /* sifting up */
307: } /* }}} */
308:
309: static spl_ptr_heap *spl_ptr_heap_clone(spl_ptr_heap *from) /* {{{ */ {
310:     int i;
311:     spl_ptr_heap *heap = emalloc(sizeof(spl_ptr_heap));
312:
313:     heap->dtor = from->dtor;
314:     heap->ctor = from->ctor;
315:     heap->cmp = from->cmp;
316:     heap->max_size = from->max_size;
317:     heap->count = from->count;
318:     heap->flags = from->flags;
319:
320:     heap->elements = safe_emalloc(sizeof(zval), from->max_size, 0);
321:     memcpy(heap->elements, from->elements, sizeof(zval) * from->max_size);
322:
323:     for (i=0; i < heap->count; ++i) {
324:         heap->elements[i] = from->elements[i];
325:     }
326:
327:     return heap;
328: } /* }}} */
329:
330:
331: static void spl_ptr_heap_destroy(spl_ptr_heap *heap) /* {{{ */ {
332:     int i;
333:
334:     for (i=0; i < heap->count; ++i) {
335:         heap->dtor(heap->elements[i]);
336:     }
337:
338:     efree(heap->elements);
339:     efree(heap);
340: } /* }}} */
341:
342:
343: static int spl_ptr_heap_count(spl_ptr_heap *heap) /* {{{ */ {
344:     return heap->count;
345: } /* }}} */
346:
347:
348: zend_object_iterator *spl_heap_get_iterator(zend_class_entry *ce, zval *obj, int by_ref) {
349:     static void spl_heap_object_free_storage(zend_object *obj) /* {{{ */ {
350:         spl_heap_object *intern = spl_heap_from_obj(obj);
351:
352:         zend_object_std_dtor(intern->std);
353:
354:         spl_ptr_heap_destroy(intern->heap);
355:     } /* }}} */
356:
357:
358: static zend_object *spl_heap_object_new_ext(zend_class_entry *class_type, zval *orig, int clone_orig) /* {{{ */ {
359:     spl_heap_object *intern;
360:     zend_class_entry *parent = class_type;
361:     int inherited = 0;
362:
363:     intern = zend_object_alloc(sizeof(spl_heap_object), parent);
364:
365:     zend_object_std_init(intern->std, class_type);
366:     object_properties_init(intern->std, class_type);
367:
368:     intern->flags = 0;
369:     intern->fptr_cmp = NULL;
370:
371:     if (orig) {
372:         spl_heap_object *other = Z_SPLHEAP_P(orig);
373:         intern->ce_get_iterator = other->ce_get_iterator;

```

```

377:
378:     IF (clone_orig) {
379:         intern->heap = spl_ptr_heap_clone(other->heap);
380:     } else {
381:         intern->heap = other->heap;
382:     }
383:
384:     intern->flags = other->flags;
385: } else {
386:     intern->heap = spl_ptr_heap_init(spl_ptr_heap_eval_max_cmp, spl_ptr_heap_eval_ctor, spl_ptr_heap_eval_dtor);
387: }
388:
389: intern->std.handlers = spl_handler_SplHeap;
390:
391: while (parent) {
392:     IF (parent == spl_ce_SplPriorityQueue) {
393:         intern->heap->cmp = spl_ptr_pqueue_eval_cmp;
394:         intern->flags = SPL_PRIORITY_EXTRA_DATA;
395:         intern->std.handlers = spl_handler_SplPriorityQueue;
396:         break;
397:     }
398:
399:     IF (parent == spl_ce_SplMinHeap) {
400:         intern->heap->cmp = spl_ptr_heap_eval_min_cmp;
401:         break;
402:     }
403:
404:     IF (parent == spl_ce_SplMaxHeap) {
405:         intern->heap->cmp = spl_ptr_heap_eval_max_cmp;
406:         break;
407:     }
408:
409:     IF (parent == spl_ce_SplHeap) {
410:         break;
411:     }
412:
413:     parent = parent->parent;
414:     inherited = 1;
415: }
416:
417: IF (!parent) { /* this must never happen */
418:     php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error, class is not child of SplHeap");
419: }
420:
421: IF (inherited) {
422:     intern->fptr_cmp = zend_hash_str_find_ptr(class_type->function_table, "compare", sizeof("compare") - 1);
423:     IF (intern->fptr_cmp->common.scope == parent) {
424:         intern->fptr_cmp = NULL;
425:     }
426:     intern->fptr_count = zend_hash_str_find_ptr(class_type->function_table, "count", sizeof("count") - 1);
427:     IF (intern->fptr_count->common.scope == parent) {
428:         intern->fptr_count = NULL;
429:     }
430: }
431:
432: return sintern->std;
433: } /* }}} */
434:
435:
436: static zend_object *spl_heap_object_new(zend_class_entry *class_type) /* {{{ */
437: {
438:     return spl_heap_object_new_ex(class_type, NULL, 0);
439: }
440: /* }}} */
441:
442: static zend_object *spl_heap_object_clone(zval *obj) /* {{{ */
443: {
444:     zend_object *old_obj;
445:     zend_object *new_obj;
446:
447:     old_obj = Z_OBJ_P(obj);
448:     new_obj = spl_heap_object_new_ex(old_obj->ce, obj, 1);
449:
450:     zend_objects_clone_members(new_obj, old_obj);
451:
452:     return new_obj;
453: }
454: /* }}} */
455:
456: static int spl_heap_object_count_elements(zval *obj, zend_long *count) /* {{{ */
457: {
458:     spl_heap_object *intern = Z_SPLHEAP_P(obj);
459:
460:     IF (intern->fptr_count) {
461:         zval rv;
462:         zend_call_method_with_0_params(object, intern->std.ce, sintern->fptr_count, "count", &rv);
463:         IF (!Z_ISUNDEF(rv)) {
464:             *count = zval_get_long(&rv);
465:             zval_ptr_dtor(&rv);
466:             return SUCCESS;
467:         }
468:         *count = 0;
469:         return FAILURE;
470:     }
471:
472:     *count = spl_ptr_heap_count(intern->heap);
473:
474:     return SUCCESS;
475: }
476: /* }}} */
477:
478: static HashTable *spl_heap_object_get_debug_info_helper(zend_class_entry *ce, zval *obj, int *is_temp) /* {{{ */
479: {
480:     spl_heap_object *intern = Z_SPLHEAP_P(obj);
481:     zval tmp_heap_array;
482:     zend_string *pnstr;
483:     HashTable *debug_info;
484:     int i;
485:
486:     *is_temp = 1;
487:
488:     IF (!intern->std.properties) {
489:         rebuild_object_properties(intern->std);
490:     }
491:
492:     debug_info = zend_new_array(zend_hash_num_elements(intern->std.properties) + 1);
493:     zend_hash_copy(debug_info, intern->std.properties, (copy_ctor_func_t) zval_add_ref);
494:
495:     pnstr = spl_gen_private_prop_name(ce, "Flags", sizeof("Flags") - 1);
496:     ZVAL_LONG(&tmp_heap_array, intern->flags);
497:     zend_hash_update(debug_info, pnstr, &tmp_heap_array);
498:     zend_string_release(pnstr);
499:
500:     pnstr = spl_gen_private_prop_name(ce, "IsCorrupted", sizeof("IsCorrupted") - 1);
501:     ZVAL_BOOL(&tmp_heap_array, intern->heap->flags & SPL_HEAP_CORRUPTED);
502:     zend_hash_update(debug_info, pnstr, &tmp_heap_array);
503:     zend_string_release(pnstr);
504:
505:     array_init(&debug_info);
506:
507:     for (i = 0; i < intern->heap->count; ++i) {
508:         add_index_zval(&debug_info, i, sintern->heap->elements[i]);
509:         IF (Z_REFCOUNTED(intern->heap->elements[i])) {
510:             Z_ADDREF(intern->heap->elements[i]);
511:         }
512:     }
513:
514:     pnstr = spl_gen_private_prop_name(ce, "Heap", sizeof("Heap") - 1);
515:     zend_hash_update(debug_info, pnstr, &debug_info);
516:     zend_string_release(pnstr);
517:
518:     return debug_info;
519: } /* }}} */
520:
521: static HashTable *spl_heap_object_get_get(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
522: {
523:     spl_heap_object *intern = Z_SPLHEAP_P(obj);
524:     *gc_data = intern->heap->elements;
525:     *gc_data_count = intern->heap->count;
526:
527:     return std_object_handlers.get_properties(obj);
528: }
529: /* }}} */
530:
531: static HashTable *spl_heap_object_get_debug_info(zval *obj, int *is_temp) /* {{{ */
532: {
533:     return spl_heap_object_get_debug_info_helper(spl_ce_SplHeap, obj, is_temp);
534: }
535: /* }}} */
536:
537: static HashTable *spl_pqueue_object_get_debug_info(zval *obj, int *is_temp) /* {{{ */
538: {
539:     return spl_heap_object_get_debug_info_helper(spl_ce_SplPriorityQueue, obj, is_temp);
540: }
541: /* }}} */
542:
543: /* {{{ proto int SplHeap::count()
544: Return the number of elements in the heap. */
545: SPL_METHOD(SplHeap, count)
546: {
547:     zend_long count;
548:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
549:
550:     IF (zend_parse_parameters_none() == FAILURE) {
551:         return;
552:     }
553:
554:     count = spl_ptr_heap_count(intern->heap);
555:     RETURN_LONG(count);
556: }
557: /* }}} */
558:
559: /* {{{ proto int SplHeap::isEmpty()
560: Return true if the heap is empty. */
561: SPL_METHOD(SplHeap, isEmpty)
562: {
563:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
564:

```

```

565:     IF (zend_parse_parameters_none() == FAILURE) {
566:         return;
567:     }
568:
569:     RETURN_BOOL(spl_ptr_heap_count(intern->heap) == 0);
570: }
571: /* }}} */
572:
573: /* {{{ proto bool SplHeap::insert(mixed value)
574: Push $value on the heap */
575: SPL_METHOD(SplHeap, insert)
576: {
577:     zval *value;
578:     spl_heap_object *intern;
579:
580:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &value) == FAILURE) {
581:         return;
582:     }
583:
584:     intern = Z_SPLHEAP_P(getThis());
585:
586:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
587:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
588:         return;
589:     }
590:
591:     Z_TRY_ADDREF_P(value);
592:     spl_ptr_heap_insert(intern->heap, value, getThis());
593:
594:     RETURN_TRUE;
595: }
596: /* }}} */
597:
598: /* {{{ proto mixed SplHeap::extract()
599: Extract the element out of the top of the heap */
600: SPL_METHOD(SplHeap, extract)
601: {
602:     spl_heap_object *intern;
603:
604:     IF (zend_parse_parameters_none() == FAILURE) {
605:         return;
606:     }
607:
608:     intern = Z_SPLHEAP_P(getThis());
609:
610:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
611:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
612:         return;
613:     }
614:
615:     spl_ptr_heap_delete_top(intern->heap, return_value, getThis());
616:
617:     IF (Z_ISUNDEF_P(return_value)) {
618:         zend_throw_exception(spl_ce_RuntimeException, "Can't extract from an empty heap", 0);
619:         return;
620:     }
621: }
622: /* }}} */
623:
624: /* {{{ proto bool SplPriorityQueue::insert(mixed value, mixed priority)
625: Push $value with the priority $priority on the priorityqueue */
626: SPL_METHOD(SplPriorityQueue, insert)
627: {
628:     zval *data, *priority, elem;
629:     spl_heap_object *intern;
630:
631:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "as", &data, &priority) == FAILURE) {
632:         return;
633:     }
634:
635:     intern = Z_SPLHEAP_P(getThis());
636:
637:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
638:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
639:         return;
640:     }
641:
642:     Z_TRY_ADDREF_P(data);
643:     Z_TRY_ADDREF_P(priority);
644:
645:     array_init(&elem);
646:     add_assoc_zval_ex(&elem, "data", sizeof("data") - 1, data);
647:     add_assoc_zval_ex(&elem, "priority", sizeof("priority") - 1, priority);
648:
649:     spl_ptr_heap_insert(intern->heap, &elem, getThis());
650:
651:     RETURN_TRUE;
652: }
653: /* }}} */
654:
655: /* {{{ proto mixed SplPriorityQueue::extract()
656: Extract the element out of the top of the priority queue */
657: SPL_METHOD(SplPriorityQueue, extract)
658: {
659:     zval value, *value_out;
660:     spl_heap_object *intern;
661:
662:     IF (zend_parse_parameters_none() == FAILURE) {
663:         return;
664:     }
665:
666:     intern = Z_SPLHEAP_P(getThis());
667:
668:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
669:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
670:         return;
671:     }
672:
673:     spl_ptr_heap_delete_top(intern->heap, &value, getThis());
674:
675:     IF (Z_ISUNDEF(value)) {
676:         zend_throw_exception(spl_ce_RuntimeException, "Can't extract from an empty heap", 0);
677:         return;
678:     }
679:
680:     value_out = spl_pqueue_extract_helper(&value, intern->flags);
681:
682:     IF (!value_out) {
683:         zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
684:         zval_ptr_dtor(&value);
685:         return;
686:     }
687:
688:     ZVAL_DEREF(value_out);
689:     ZVAL_COPY(&return_value, value_out);
690:     zval_ptr_dtor(&value);
691: }
692: /* }}} */
693:
694: /* {{{ proto mixed SplPriorityQueue::top()
695: Peek at the top element of the priority queue */
696: SPL_METHOD(SplPriorityQueue, top)
697: {
698:     zval *value, *value_out;
699:     spl_heap_object *intern;
700:
701:     IF (zend_parse_parameters_none() == FAILURE) {
702:         return;
703:     }
704:
705:     intern = Z_SPLHEAP_P(getThis());
706:
707:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
708:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
709:         return;
710:     }
711:
712:     value = spl_ptr_heap_top(intern->heap);
713:
714:     IF (!value) {
715:         zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty heap", 0);
716:         return;
717:     }
718:
719:     value_out = spl_pqueue_extract_helper(value, intern->flags);
720:
721:     IF (!value_out) {
722:         zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
723:         return;
724:     }
725:
726:     ZVAL_DEREF(value_out);
727:     ZVAL_COPY(&return_value, value_out);
728: }
729: /* }}} */
730:
731: /* {{{ proto int SplPriorityQueue::isotExtractFlags(int flags)
732: Get the flags of extraction */
733: SPL_METHOD(SplPriorityQueue, isotExtractFlags)
734: {
735:     zend_long value;
736:     spl_heap_object *intern;
737:
738:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "i", &value) == FAILURE) {
739:         return;
740:     }
741:
742:     intern = Z_SPLHEAP_P(getThis());
743:
744:     intern->flags = value & SPL_PRIORITY_EXTRA_MASK;
745:
746:     RETURN_LONG(intern->flags);
747: }
748: /* }}} */
749:
750: /* {{{ proto int SplPriorityQueue::getExtractFlags()
751: Get the flags of extraction */
752:

```

```

753: SPL_METHOD(SplPriorityQueue, getExtractFlags)
754: {
755:     spl_heap_object *intern;
756:
757:     IF (zend_parse_parameters_none() == FAILURE) {
758:         return;
759:     }
760:
761:     intern = Z_SPLHEAP_P(getThis());
762:
763:     RETURN_LONG(intern->flags);
764: }
765: /* }}} */
766:
767: /* {{{ proto int SplHeap::recoverFromCorruption()
768:  * Recover from a corrupted state */
769: SPL_METHOD(SplHeap, recoverFromCorruption)
770: {
771:     spl_heap_object *intern;
772:
773:     IF (zend_parse_parameters_none() == FAILURE) {
774:         return;
775:     }
776:
777:     intern = Z_SPLHEAP_P(getThis());
778:
779:     intern->heap->flags = intern->heap->flags & SPL_HEAP_CORRUPTED;
780:
781:     RETURN_TRUE;
782: }
783: /* }}} */
784:
785: /* {{{ proto int SplHeap::isCorrupted()
786:  * Tells if the heap is in a corrupted state */
787: SPL_METHOD(SplHeap, isCorrupted)
788: {
789:     spl_heap_object *intern;
790:
791:     IF (zend_parse_parameters_none() == FAILURE) {
792:         return;
793:     }
794:
795:     intern = Z_SPLHEAP_P(getThis());
796:
797:     RETURN_BOOL(intern->heap->flags & SPL_HEAP_CORRUPTED);
798: }
799: /* }}} */
800:
801: /* {{{ proto bool SplPriorityQueue::compare(mixed $a, mixed $b)
802:  * compare the priorities */
803: SPL_METHOD(SplPriorityQueue, compare)
804: {
805:     zval *a, *b;
806:
807:     IF (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "zz", &a, &b) == FAILURE) {
808:         return;
809:     }
810:
811:     RETURN_LONG(spl_ptr_heap_zval_max_cmp(a, b, NULL));
812: }
813: /* }}} */
814:
815: /* {{{ proto mixed SplHeap::top()
816:  * Peek at the top element of the heap */
817: SPL_METHOD(SplHeap, top)
818: {
819:     zval *value;
820:     spl_heap_object *intern;
821:
822:     IF (zend_parse_parameters_none() == FAILURE) {
823:         return;
824:     }
825:
826:     intern = Z_SPLHEAP_P(getThis());
827:
828:     IF (intern->heap->flags & SPL_HEAP_CORRUPTED) {
829:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
830:         return;
831:     }
832:
833:     value = spl_ptr_heap_top(intern->heap);
834:
835:     IF (!value) {
836:         zend_throw_exception(spl_ce_RuntimeException, "Can't peek at an empty heap", 0);
837:         return;
838:     }
839:
840:     ZVAL_DEREF(value);
841:     ZVAL_COPY(return_value, value);
842: }
843: /* }}} */
844:
845: /* {{{ proto bool SplMinHeap::compare(mixed $a, mixed $b)
846:  * compare the values */
847: SPL_METHOD(SplMinHeap, compare)
848: {
849:     zval *a, *b;
850:
851:     IF (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "zz", &a, &b) == FAILURE) {
852:         return;
853:     }
854:
855:     RETURN_LONG(spl_ptr_heap_zval_min_cmp(a, b, NULL));
856: }
857: /* }}} */
858:
859: /* {{{ proto bool SplMaxHeap::compare(mixed $a, mixed $b)
860:  * compare the values */
861: SPL_METHOD(SplMaxHeap, compare)
862: {
863:     zval *a, *b;
864:
865:     IF (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "zz", &a, &b) == FAILURE) {
866:         return;
867:     }
868:
869:     RETURN_LONG(spl_ptr_heap_zval_max_cmp(a, b, NULL));
870: }
871: /* }}} */
872:
873: static void spl_heap_it_dtor(zend_object_iterator *iter) /* {{{ */
874: {
875:     spl_heap_it *iterator = (spl_heap_it *)iter;
876:
877:     zend_user_it_invalidata_current(iter);
878:     zval_ptr_dtor(iterator->intern.it.data);
879: }
880: /* }}} */
881:
882: static void spl_heap_it_rewind(zend_object_iterator *iter) /* {{{ */
883: {
884:     /* do nothing, the iterator always points to the top element */
885: }
886: /* }}} */
887:
888: static int spl_heap_it_valid(zend_object_iterator *iter) /* {{{ */
889: {
890:     return ((Z_SPLHEAP_P(iter->data)->heap->count != 0 ? SUCCESS : FAILURE);
891: }
892: /* }}} */
893:
894: static zval *spl_heap_it_get_current_data(zend_object_iterator *iter) /* {{{ */
895: {
896:     spl_heap_object *object = Z_SPLHEAP_P(iter->data);
897:     zval *element = object->heap->elements[0];
898:
899:     IF (object->heap->flags & SPL_HEAP_CORRUPTED) {
900:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
901:         return NULL;
902:     }
903:
904:     IF (object->heap->count == 0 || Z_UNDEF_P(element)) {
905:         return NULL;
906:     } else {
907:         return element;
908:     }
909: }
910: /* }}} */
911:
912: static zval *spl_pqueue_it_get_current_data(zend_object_iterator *iter) /* {{{ */
913: {
914:     spl_heap_object *object = Z_SPLHEAP_P(iter->data);
915:     zval *element = object->heap->elements[0];
916:
917:     IF (object->heap->flags & SPL_HEAP_CORRUPTED) {
918:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
919:         return NULL;
920:     }
921:
922:     IF (object->heap->count == 0 || Z_UNDEF_P(element)) {
923:         return NULL;
924:     } else {
925:         zval *data = spl_pqueue_extract_helper(element, object->flags);
926:         IF (!data) {
927:             zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
928:         }
929:         return data;
930:     }
931: }
932: /* }}} */
933:
934: static void spl_heap_it_get_current_key(zend_object_iterator *iter, zval **key) /* {{{ */
935: {
936:     spl_heap_object *object = Z_SPLHEAP_P(iter->data);
937:
938:     ZVAL_LONG(key, object->heap->count - 1);
939: }
940: /* }}} */

```

```

941:
942: static void spl_heap_it_move_forward(zend_object_iterator *iter) /* {{{ */
943: {
944:     spl_heap_object *object = Z_SPLHEAP_P(iter->data);
945:     zval elem;
946:
947:     IF (object->heap->flags & SPL_HEAP_CORRUPTED) {
948:         zend_throw_exception(spl_ce_RuntimeException, "Heap is corrupted, heap properties are no longer ensured.", 0);
949:         return;
950:     }
951:
952:     spl_ptr_heap_delete_top(object->heap, elem, iter->data);
953:
954:     zval_ptr_dtor(&elem);
955:
956:     zend_user_it_invalidata_current(iter);
957: }
958: /* }}} */
959:
960: /* {{{ proto int SplHeap::key()
961:  * Return current array key */
962: SPL_METHOD(SplHeap, key)
963: {
964:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
965:
966:     IF (zend_parse_parameters_none() == FAILURE) {
967:         return;
968:     }
969:
970:     RETURN_LONG(intern->heap->count - 1);
971: }
972: /* }}} */
973:
974: /* {{{ proto void SplHeap::next()
975:  * Move to next entry */
976: SPL_METHOD(SplHeap, next)
977: {
978:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
979:     zval elem;
980:     spl_ptr_heap_delete_top(intern->heap, elem, getThis());
981:
982:     IF (zend_parse_parameters_none() == FAILURE) {
983:         return;
984:     }
985:
986:     zval_ptr_dtor(&elem);
987: }
988: /* }}} */
989:
990: /* {{{ proto bool SplHeap::valid()
991:  * Check whether the datastructure contains more entries */
992: SPL_METHOD(SplHeap, valid)
993: {
994:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
995:
996:     IF (zend_parse_parameters_none() == FAILURE) {
997:         return;
998:     }
999:
1000:     RETURN_BOOL(intern->heap->count != 0);
1001: }
1002: /* }}} */
1003:
1004: /* {{{ proto void SplHeap::rewind()
1005:  * Rewind the datastructure back to the start */
1006: SPL_METHOD(SplHeap, rewind)
1007: {
1008:     IF (zend_parse_parameters_none() == FAILURE) {
1009:         return;
1010:     }
1011:     /* do nothing, the iterator always points to the top element */
1012: }
1013: /* }}} */
1014:
1015: /* {{{ proto mixed NULL SplHeap::current()
1016:  * Return current datastructure entry */
1017: SPL_METHOD(SplHeap, current)
1018: {
1019:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
1020:     zval *element = intern->heap->elements[0];
1021:
1022:     IF (zend_parse_parameters_none() == FAILURE) {
1023:         return;
1024:     }
1025:
1026:     IF (!intern->heap->count || Z_UNDEF_P(element)) {
1027:         RETURN_NULL();
1028:     } else {
1029:         ZVAL_DEREF(element);
1030:         ZVAL_COPY(return_value, element);
1031:     }
1032: }
1033: /* }}} */
1034:
1035: /* {{{ proto mixed NULL SplPriorityQueue::current()
1036:  * Return current datastructure entry */
1037: SPL_METHOD(SplPriorityQueue, current)
1038: {
1039:     spl_heap_object *intern = Z_SPLHEAP_P(getThis());
1040:     zval *element = intern->heap->elements[0];
1041:
1042:     IF (zend_parse_parameters_none() == FAILURE) {
1043:         return;
1044:     }
1045:
1046:     IF (!intern->heap->count || Z_UNDEF_P(element)) {
1047:         RETURN_NULL();
1048:     } else {
1049:         zval *data = spl_pqueue_extract_helper(element, intern->flags);
1050:
1051:         IF (!data) {
1052:             zend_error(E_RECOVERABLE_ERROR, "Unable to extract from the PriorityQueue node");
1053:             RETURN_NULL();
1054:         }
1055:
1056:         ZVAL_DEREF(data);
1057:         ZVAL_COPY(return_value, data);
1058:     }
1059: }
1060: /* }}} */
1061:
1062: /* Iterator handler table */
1063: static const zend_object_iterator_funcs spl_heap_it_funcs = {
1064:     spl_heap_it_dtor,
1065:     spl_heap_it_valid,
1066:     spl_heap_it_get_current_data,
1067:     spl_heap_it_get_current_key,
1068:     spl_heap_it_move_forward,
1069:     spl_heap_it_rewind,
1070:     NULL
1071: };
1072:
1073: static const zend_object_iterator_funcs spl_pqueue_it_funcs = {
1074:     spl_heap_it_dtor,
1075:     spl_heap_it_valid,
1076:     spl_pqueue_it_get_current_data,
1077:     spl_heap_it_get_current_key,
1078:     spl_heap_it_move_forward,
1079:     spl_heap_it_rewind,
1080:     NULL
1081: };
1082:
1083: zend_object_iterator *spl_heap_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1084: {
1085:     spl_heap_it *iterator;
1086:     spl_heap_object *heap_object = Z_SPLHEAP_P(object);
1087:
1088:     IF (by_ref) {
1089:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1090:         return NULL;
1091:     }
1092:
1093:     iterator = emalloc(sizeof(spl_heap_it));
1094:
1095:     zend_iterator_init(iterator->intern.it);
1096:
1097:     ZVAL_COPY(iterator->intern.it.data, object);
1098:     iterator->intern.it.funcs = &spl_heap_it_funcs;
1099:     iterator->intern.ce = ce;
1100:     iterator->flags = heap_object->flags;
1101:     ZVAL_UNDEF(iterator->intern.value);
1102:
1103:     return iterator->intern.it;
1104: }
1105: /* }}} */
1106:
1107: zend_object_iterator *spl_pqueue_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1108: {
1109:     spl_heap_it *iterator;
1110:     spl_heap_object *heap_object = Z_SPLHEAP_P(object);
1111:
1112:     IF (by_ref) {
1113:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1114:         return NULL;
1115:     }
1116:
1117:     iterator = emalloc(sizeof(spl_heap_it));
1118:
1119:     zend_iterator_init((zend_object_iterator *)iterator);
1120:
1121:     ZVAL_COPY(iterator->intern.it.data, object);
1122:     iterator->intern.it.funcs = &spl_pqueue_it_funcs;
1123:     iterator->intern.ce = ce;
1124:     iterator->flags = heap_object->flags;
1125:
1126:     ZVAL_UNDEF(iterator->intern.value);
1127:
1128:     return iterator->intern.it;

```

```
1129: }
1130: /* }}} */
1131:
1132: ZEND_BEGIN_ARG_INFO(arginfo_heap_insert, 0)
1133: ZEND_ARG_INFO(0, value)
1134: ZEND_END_ARG_INFO()
1135:
1136: ZEND_BEGIN_ARG_INFO(arginfo_heap_compare, 0)
1137: ZEND_ARG_INFO(0, a)
1138: ZEND_ARG_INFO(0, b)
1139: ZEND_END_ARG_INFO()
1140:
1141: ZEND_BEGIN_ARG_INFO(arginfo_pqqueue_insert, 0)
1142: ZEND_ARG_INFO(0, value)
1143: ZEND_ARG_INFO(0, priority)
1144: ZEND_END_ARG_INFO()
1145:
1146: ZEND_BEGIN_ARG_INFO(arginfo_pqqueue_setflags, 0)
1147: ZEND_ARG_INFO(0, flags)
1148: ZEND_END_ARG_INFO()
1149:
1150: ZEND_BEGIN_ARG_INFO(arginfo_splheap_void, 0)
1151: ZEND_END_ARG_INFO()
1152:
1153: static const zend_function_entry spl_funcs_SplMinHeap[] = {
1154:     SPL_ME(SplMinHeap, compare, arginfo_heap_compare, ZEND_ACC_PROTECTED)
1155:     PHP_FE_END
1156: };
1157:
1158: static const zend_function_entry spl_funcs_SplMaxHeap[] = {
1159:     SPL_ME(SplMaxHeap, compare, arginfo_heap_compare, ZEND_ACC_PROTECTED)
1160:     PHP_FE_END
1161: };
1162:
1163: static const zend_function_entry spl_funcs_SplPriorityQueue[] = {
1164:     SPL_ME(SplPriorityQueue, compare, arginfo_heap_compare, ZEND_ACC_PUBLIC)
1165:     SPL_ME(SplPriorityQueue, insert, arginfo_pqqueue_insert, ZEND_ACC_PUBLIC)
1166:     SPL_ME(SplPriorityQueue, setExtractFlags, arginfo_pqqueue_setflags, ZEND_ACC_PUBLIC)
1167:     SPL_ME(SplPriorityQueue, getExtractFlags, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1168:     SPL_ME(SplPriorityQueue, top, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1169:     SPL_ME(SplPriorityQueue, extract, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1170:     SPL_ME(SplHeap, count, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1171:     SPL_ME(SplHeap, isEmpty, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1172:     SPL_ME(SplPriorityQueue, current, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1173:     SPL_ME(SplHeap, key, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1174:     SPL_ME(SplHeap, rewind, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1175:     SPL_ME(SplHeap, valid, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1176:     SPL_ME(SplHeap, recoverFromCorruption, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1177:     SPL_ME(SplHeap, isCorrupted, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1178:     PHP_FE_END
1179: };
1180:
1181: static const zend_function_entry spl_funcs_SplHeap[] = {
1182:     SPL_ME(SplHeap, extract, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1183:     SPL_ME(SplHeap, insert, arginfo_heap_insert, ZEND_ACC_PUBLIC)
1184:     SPL_ME(SplHeap, top, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1185:     SPL_ME(SplHeap, count, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1186:     SPL_ME(SplHeap, isEmpty, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1187:     SPL_ME(SplHeap, rewind, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1188:     SPL_ME(SplHeap, current, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1189:     SPL_ME(SplHeap, key, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1190:     SPL_ME(SplHeap, next, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1191:     SPL_ME(SplHeap, valid, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1192:     SPL_ME(SplHeap, recoverFromCorruption, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1193:     SPL_ME(SplHeap, isCorrupted, arginfo_splheap_void, ZEND_ACC_PUBLIC)
1194:     ZEND_FENTRY(compare, NULL, NULL, ZEND_ACC_PROTECTED|ZEND_ACC_ABSTRACT)
1195:     PHP_FE_END
1196: };
1197: /* }}} */
1198:
1199: PHP_MINIT_FUNCTION(spl_heap) /* {{{ */
1200: {
1201:     REGISTER_SPL_STD_CLASS_EX(SplHeap, spl_heap_object_new, spl_funcs_SplHeap);
1202:     memcpy(&spl_handler_SplHeap, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1203:
1204:     spl_handler_SplHeap.offset = XOffsetOf(spl_heap_object, std);
1205:     spl_handler_SplHeap.clone_obj = spl_heap_object_clone;
1206:     spl_handler_SplHeap.count_elements = spl_heap_object_count_elements;
1207:     spl_handler_SplHeap.get_debug_info = spl_heap_object_get_debug_info;
1208:     spl_handler_SplHeap.get_gc = spl_heap_object_get_gc;
1209:     spl_handler_SplHeap.dtor_obj = zend_objects_destroy_object;
1210:     spl_handler_SplHeap.free_obj = spl_heap_object_free_storage;
1211:
1212:     REGISTER_SPL_IMPLMENTS(SplHeap, Iterator);
1213:     REGISTER_SPL_IMPLMENTS(SplHeap, Countable);
1214:
1215:     spl_ow_SplHeap->get_iterator = spl_heap_get_iterator;
1216:
1217:     REGISTER_SPL_SUB_CLASS_EX(SplMinHeap, SplHeap, spl_heap_object_new, spl_funcs_SplMinHeap);
1218:     REGISTER_SPL_SUB_CLASS_EX(SplMaxHeap, SplHeap, spl_heap_object_new, spl_funcs_SplMaxHeap);
1219:
1220:     spl_ow_SplMaxHeap->get_iterator = spl_heap_get_iterator;
1221:     spl_ow_SplMinHeap->get_iterator = spl_heap_get_iterator;
1222:
1223:     REGISTER_SPL_STD_CLASS_EX(SplPriorityQueue, spl_heap_object_new, spl_funcs_SplPriorityQueue);
1224:     memcpy(&spl_handler_SplPriorityQueue, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1225:
1226:     spl_handler_SplPriorityQueue.offset = XOffsetOf(spl_heap_object, std);
1227:     spl_handler_SplPriorityQueue.clone_obj = spl_heap_object_clone;
1228:     spl_handler_SplPriorityQueue.count_elements = spl_heap_object_count_elements;
1229:     spl_handler_SplPriorityQueue.get_debug_info = spl_pqqueue_object_get_debug_info;
1230:     spl_handler_SplPriorityQueue.get_gc = spl_heap_object_get_gc;
1231:     spl_handler_SplPriorityQueue.dtor_obj = zend_objects_destroy_object;
1232:     spl_handler_SplPriorityQueue.free_obj = spl_heap_object_free_storage;
1233:
1234:     REGISTER_SPL_IMPLMENTS(SplPriorityQueue, Iterator);
1235:     REGISTER_SPL_IMPLMENTS(SplPriorityQueue, Countable);
1236:
1237:     spl_ow_SplPriorityQueue->get_iterator = spl_pqqueue_get_iterator;
1238:
1239:     REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_BOTH", SPL_PQEXTR_EXTR_BOTH);
1240:     REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_PRIORITY", SPL_PQEXTR_EXTR_PRIORITY);
1241:     REGISTER_SPL_CLASS_CONST_LONG(SplPriorityQueue, "EXTR_DATA", SPL_PQEXTR_EXTR_DATA);
1242:
1243:     return SUCCESS;
1244: }
1245: /* }}} */
1246:
1247: /*
1248:  * Local variables:
1249:  * tab-width: 4
1250:  * c-basic-offset: 4
1251:  * End:
1252:  * vim600: fdm=marker
1253:  * vim: noet sw=4 ts=4
1254:  */
1255: }
```

```

1: 21: 22: 23: 24: 25: 26: 27: 28: 29: 30: 31: 32: 33: 34: 35: 36: 37: 38: 39: 40: 41: 42: 43: 44: 45: 46: 47: 48: 49: 50: 51: 52: 53: 54: 55: 56: 57: 58: 59: 60: 61: 62: 63: 64: 65: 66: 67: 68: 69: 70: 71: 72: 73: 74: 75: 76: 77: 78: 79: 80: 81: 82: 83: 84: 85: 86: 87: 88: 89: 90: 91: 92: 93: 94: 95: 96: 97: 98: 99: 100: 101: 102: 103: 104: 105: 106: 107: 108: 109: 110: 111: 112: 113: 114: 115: 116: 117: 118: 119: 120: 121: 122: 123: 124: 125: 126: 127: 128: 129: 130: 131: 132: 133: 134: 135: 136: 137: 138: 139: 140: 141: 142: 143: 144: 145: 146: 147: 148: 149: 150: 151: 152: 153: 154: 155: 156: 157: 158: 159: 160: 161: 162: 163: 164: 165: 166: 167: 168: 169: 170: 171: 172: 173: 174: 175: 176: 177: 178: 179: 180: 181: 182: 183: 184: 185: 186: 187: 188: 189: 190: 191: 192: 193: 194: 195: 196: 197: 198: 199: 200: 201: 202: 203: 204: 205: 206: 207: 208: 209: 210: 211: 212: 213: 214: 215: 216: 217: 218: 219: 220: 221: 222: 223: 224: 225: 226: 227: 228: 229: 230: 231: 232: 233: 234: 235: 236: 237: 238: 239: 240: 241: 242: 243: 244: 245: 246: 247: 248: 249: 250: 251: 252: 253: 254: 255: 256: 257: 258: 259: 260: 261: 262: 263: 264: 265: 266: 267: 268: 269: 270: 271: 272: 273: 274: 275: 276: 277: 278: 279: 280: 281: 282: 283: 284: 285: 286: 287: 288: 289: 290: 291: 292: 293: 294: 295: 296: 297: 298: 299: 300: 301: 302: 303: 304: 305: 306: 307: 308: 309: 310: 311: 312: 313: 314: 315: 316: 317: 318: 319: 320: 321: 322: 323: 324: 325: 326: 327: 328: 329: 330: 331: 332: 333: 334: 335: 336: 337: 338: 339: 340: 341: 342: 343: 344: 345: 346: 347: 348: 349: 350: 351: 352: 353: 354: 355: 356: 357: 358: 359: 360: 361: 362: 363: 364: 365: 366: 367: 368: 369: 370: 371: 372: 373: 374: 375: 376: 377: 378: 379: 380: 381: 382: 383: 384: 385: 386: 387: 388: 389: 390: 391: 392: 393: 394: 395: 396: 397: 398: 399: 400: 401: 402: 403: 404: 405: 406: 407: 408: 409: 410: 411: 412: 413: 414: 415: 416: 417: 418: 419: 420: 421: 422: 423: 424: 425: 426: 427: 428: 429: 430: 431: 432: 433: 434: 435: 436: 437: 438: 439: 440: 441: 442: 443: 444: 445: 446: 447: 448: 449: 450: 451: 452: 453: 454: 455: 456: 457: 458: 459: 460: 461: 462: 463: 464: 465: 466: 467: 468: 469: 470: 471: 472: 473: 474: 475: 476: 477: 478: 479: 480: 481: 482: 483: 484: 485: 486: 487: 488: 489: 490: 491: 492: 493: 494: 495: 496: 497: 498: 499: 500: 501: 502: 503: 504: 505: 506: 507: 508: 509: 510: 511: 512: 513: 514: 515: 516: 517: 518: 519: 520: 521: 522: 523: 524: 525: 526: 527: 528: 529: 530: 531: 532: 533: 534: 535: 536: 537: 538: 539: 540: 541: 542: 543: 544: 545: 546: 547: 548: 549: 550: 551: 552: 553: 554: 555: 556: 557: 558: 559: 560: 561: 562: 563: 564: 565: 566: 567: 568: 569: 570: 571: 572: 573: 574: 575: 576: 577: 578: 579: 580: 581: 582: 583: 584: 585: 586: 587: 588: 589: 590: 591: 592: 593: 594: 595: 596: 597: 598: 599: 600: 601: 602: 603: 604: 605: 606: 607: 608: 609: 610: 611: 612: 613: 614: 615: 616: 617: 618: 619: 620: 621: 622: 623: 624: 625: 626: 627: 628: 629: 630: 631: 632: 633: 634: 635: 636: 637: 638: 639: 640: 641: 642: 643: 644: 645: 646: 647: 648: 649: 650: 651: 652: 653: 654: 655: 656: 657: 658: 659: 660: 661: 662: 663: 664: 665: 666: 667: 668: 669: 670: 671: 672: 673: 674: 675: 676: 677: 678: 679: 680: 681: 682: 683: 684: 685: 686: 687: 688: 689: 690: 691: 692: 693: 694: 695: 696: 697: 698: 699: 700: 701: 702: 703: 704: 705: 706: 707: 708: 709: 710: 711: 712: 713: 714: 715: 716: 717: 718: 719: 720: 721: 722: 723: 724: 725: 726: 727: 728: 729: 730: 731: 732: 733: 734: 735: 736: 737: 738: 739: 740: 741: 742: 743: 744: 745: 746: 747: 748: 749: 750: 751: 752: 753: 754: 755: 756: 757: 758: 759: 760: 761: 762: 763: 764: 765: 766: 767: 768: 769: 770: 771: 772: 773: 774: 775: 776: 777: 778: 779: 780: 781: 782: 783: 784: 785: 786: 787: 788: 789: 790: 791: 792: 793: 794: 795: 796: 797: 798: 799: 800: 801: 802: 803: 804: 805: 806: 807: 808: 809: 810: 811: 812: 813: 814: 815: 816: 817: 818: 819: 820: 821: 822: 823: 824: 825: 826: 827: 828: 829: 830: 831: 832: 833: 834: 835: 836: 837: 838: 839: 840: 841: 842: 843: 844: 845: 846: 847: 848: 849: 850: 851: 852: 853: 854:
```

```

377: return spl_object_storage_new_ex(class_type, NULL);
378: }
379: /* }}} */
380:
381: int spl_object_storage_contains(spl_SplObjectStorage *intern, zval *this, zval *obj) /* {{{ */
382: {
383:     int found;
384:     zend_hash_key_key;
385:     if (spl_object_storage_get_hash(key, intern, this, obj) == FAILURE) {
386:         return 0;
387:     }
388:     if (key.key) {
389:         found = zend_hash_exists(intern->storage, key.key);
390:     } else {
391:         found = zend_hash_index_exists(intern->storage, key.h);
392:     }
393:     if (found) {
394:         spl_object_storage_free_hash(intern, &key);
395:         return found;
396:     } /* }}} */
397:
398: /* {{{ proto void SplObjectStorage::attach(object obj, mixed inf = NULL)
399:  Attaches an object to the storage if not yet contained */
400: SPL_METHOD(spl_object_storage_attach)
401: {
402:     zval *obj, *inf = NULL;
403:
404:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
405:
406:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o!i", &obj, &inf) == FAILURE) {
407:         return;
408:     }
409:     spl_object_storage_attach(intern, getThis(), obj, inf);
410: } /* }}} */
411:
412: /* {{{ proto void SplObjectStorage::detach(object obj)
413:  Detaches an object from the storage */
414: SPL_METHOD(spl_object_storage_detach)
415: {
416:     zval *obj;
417:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
418:
419:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
420:         return;
421:     }
422:     spl_object_storage_detach(intern, getThis(), obj);
423:
424:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
425:     intern->index = 0;
426: } /* }}} */
427:
428: /* {{{ proto string SplObjectStorage::getHash(object obj)
429:  Returns the hash of an object */
430: SPL_METHOD(spl_object_storage_getHash)
431: {
432:     zval *obj;
433:
434:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
435:         return;
436:     }
437:
438:     RETURN_NEW_STR(PHP_SPL_OBJECT_HASH(obj));
439: } /* }}} */
440:
441: /* {{{ proto mixed SplObjectStorage::offsetGet(object obj)
442:  Returns associated information for a stored object */
443: SPL_METHOD(spl_object_storage_offsetGet)
444: {
445:     zval *obj;
446:     spl_SplObjectStorageElement *element;
447:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
448:     zend_hash_key_key;
449:
450:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
451:         return;
452:     }
453:
454:     if (spl_object_storage_get_hash(key, intern, getThis(), obj) == FAILURE) {
455:         return;
456:     }
457:
458:     element = spl_object_storage_get(intern, &key);
459:     spl_object_storage_free_hash(intern, &key);
460:
461:     if (!element) {
462:         zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Object not found");
463:     } else {
464:         zval *value = element->inf;
465:
466:         ZVAL_DEREF(value);
467:         ZVAL_COPY(&return_value, value);
468:     }
469: } /* }}} */
470:
471: /* {{{ proto bool SplObjectStorage::addAll(SplObjectStorage $os)
472:  Add all elements contained in $os */
473: SPL_METHOD(spl_object_storage_addAll)
474: {
475:     zval *obj;
476:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
477:     spl_SplObjectStorage *other;
478:
479:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, &spl_ce_SplObjectStorage) == FAILURE) {
480:         return;
481:     }
482:     other = Z_SPLOBJSTORAGE_P(obj);
483:
484:     spl_object_storage_addAll(intern, getThis(), other);
485:
486:     RETURN_LONG(zend_hash_num_elements(intern->storage));
487: } /* }}} */
488:
489: /* {{{ proto bool SplObjectStorage::removeAll(SplObjectStorage $os)
490:  Remove all elements contained in $os */
491: SPL_METHOD(spl_object_storage_removeAll)
492: {
493:     zval *obj;
494:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
495:     spl_SplObjectStorage *other;
496:     spl_SplObjectStorageElement *element;
497:
498:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, &spl_ce_SplObjectStorage) == FAILURE) {
499:         return;
500:     }
501:     other = Z_SPLOBJSTORAGE_P(obj);
502:
503:     zend_hash_internal_pointer_reset_ex(other->storage);
504:     while ((element = zend_hash_get_current_data_ptr_ex(other->storage)) != NULL) {
505:         if (spl_object_storage_detach(intern, getThis(), element->obj) == FAILURE) {
506:             spl_hash_move_forward(other->storage);
507:         }
508:     }
509:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
510:     intern->index = 0;
511:     RETURN_LONG(zend_hash_num_elements(intern->storage));
512: } /* }}} */
513:
514: /* {{{ proto bool SplObjectStorage::removeAllExcept(SplObjectStorage $os)
515:  Remove elements not common to both this SplObjectStorage instance and $os */
516: SPL_METHOD(spl_object_storage_removeAllExcept)
517: {
518:     zval *obj;
519:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
520:     spl_SplObjectStorage *other;
521:     spl_SplObjectStorageElement *element;
522:
523:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "O", &obj, &spl_ce_SplObjectStorage) == FAILURE) {
524:         return;
525:     }
526:     other = Z_SPLOBJSTORAGE_P(obj);
527:
528:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
529:     while ((element = zend_hash_get_current_data_ptr_ex(intern->storage)) != NULL) {
530:         if (!spl_object_storage_contains(other, getThis(), element->obj)) {
531:             spl_object_storage_detach(intern, getThis(), element->obj);
532:         }
533:     }
534:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
535:     intern->index = 0;
536:     RETURN_LONG(zend_hash_num_elements(intern->storage));
537: } /* }}} */
538:
539: /* {{{ proto bool SplObjectStorage::contains(object obj)
540:  Determine whether an object is contained in the storage */
541: SPL_METHOD(spl_object_storage_contains)
542: {
543:     zval *obj;
544:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
545:
546:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
547:         return;
548:     }
549:
550:     RETURN_BOOL(spl_object_storage_contains(intern, getThis(), obj));
551: } /* }}} */
552:
553: /* {{{ proto int SplObjectStorage::count()
554:  Determine number of objects in storage */
555: SPL_METHOD(spl_object_storage_count)
556: {
557:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
558:
559:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "o", &obj) == FAILURE) {
560:         return;
561:     }
562:
563:     RETURN_LONG(zend_hash_num_elements(intern->storage));
564: } /* }}} */
565:
566: zend_long mode = COUNT_NORMAL;
567:
568: if (zend_parse_parameters(ZEND_NUM_ARGS(), "i!", &mode) == FAILURE) {
569:     return;
570: }
571:
572: if (mode == COUNT_RECURSIVE) {
573:     zend_long ret;
574:
575:     if (mode != COUNT_RECURSIVE) {
576:         ret = zend_hash_num_elements(intern->storage);
577:     } else {
578:         ret = php_count_recursive(intern->storage);
579:     }
580:     RETURN_LONG(ret);
581: }
582:
583: RETURN_LONG(zend_hash_num_elements(intern->storage));
584: } /* }}} */
585:
586: /* {{{ proto void SplObjectStorage::rewind()
587:  Rewind to first position */
588: SPL_METHOD(spl_object_storage_rewind)
589: {
590:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
591:
592:     if (zend_parse_parameters_none() == FAILURE) {
593:         return;
594:     }
595:
596:     zend_hash_internal_pointer_reset_ex(intern->storage, &intern->pos);
597:     intern->index = 0;
598: } /* }}} */
599:
600: /* {{{ proto bool SplObjectStorage::valid()
601:  Returns whether current position is valid */
602: SPL_METHOD(spl_object_storage_valid)
603: {
604:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
605:
606:     if (zend_parse_parameters_none() == FAILURE) {
607:         return;
608:     }
609:
610:     RETURN_BOOL(zend_hash_has_more_elements_ex(intern->storage, &intern->pos) == SUCCESS);
611: } /* }}} */
612:
613: /* {{{ proto mixed SplObjectStorage::key()
614:  Returns current key */
615: SPL_METHOD(spl_object_storage_key)
616: {
617:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
618:
619:     if (zend_parse_parameters_none() == FAILURE) {
620:         return;
621:     }
622:
623:     RETURN_LONG(intern->index);
624: } /* }}} */
625:
626: /* {{{ proto mixed SplObjectStorage::current()
627:  Returns current element */
628: SPL_METHOD(spl_object_storage_current)
629: {
630:     spl_SplObjectStorageElement *element;
631:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
632:
633:     if (zend_parse_parameters_none() == FAILURE) {
634:         return;
635:     }
636:
637:     if ((element = zend_hash_get_current_data_ptr_ex(intern->storage, &intern->pos)) == NULL) {
638:         return;
639:     }
640:     ZVAL_COPY(&return_value, element->obj);
641: } /* }}} */
642:
643: /* {{{ proto mixed SplObjectStorage::getInfo()
644:  Returns associated information to current element */
645: SPL_METHOD(spl_object_storage_getInfo)
646: {
647:     spl_SplObjectStorageElement *element;
648:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
649:     zval *inf;
650:
651:     if (zend_parse_parameters_none() == FAILURE) {
652:         return;
653:     }
654:
655:     if ((element = zend_hash_get_current_data_ptr_ex(intern->storage, &intern->pos)) == NULL) {
656:         return;
657:     }
658:     ZVAL_COPY(&return_value, element->inf);
659: } /* }}} */
660:
661: /* {{{ proto mixed SplObjectStorage::setInfo(mixed $inf)
662:  Sets associated information of current element to $inf */
663: SPL_METHOD(spl_object_storage_setInfo)
664: {
665:     spl_SplObjectStorageElement *element;
666:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
667:     zval *inf;
668:
669:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &inf) == FAILURE) {
670:         return;
671:     }
672:
673:     if ((element = zend_hash_get_current_data_ptr_ex(intern->storage, &intern->pos)) == NULL) {
674:         return;
675:     }
676:     zval *ptr = element->inf;
677:     ZVAL_COPY(&ptr, &inf);
678: } /* }}} */
679:
680: /* {{{ proto void SplObjectStorage::next()
681:  Moves position forward */
682: SPL_METHOD(spl_object_storage_next)
683: {
684:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
685:
686:     if (zend_parse_parameters_none() == FAILURE) {
687:         return;
688:     }
689:
690:     zend_hash_move_forward_ex(intern->storage, &intern->pos);
691:     intern->index++;
692: } /* }}} */
693:
694: /* {{{ proto string SplObjectStorage::serialize()
695:  Serializes storage */
696: SPL_METHOD(spl_object_storage_serialize)
697: {
698:     spl_SplObjectStorage *intern = Z_SPLOBJSTORAGE_P(getThis());
699:
700:     spl_SplObjectStorageElement *element;
701:     zval members, flags;
702:     HashPosition pos;
703:     PHP_SERIALIZE_DATA var_hash;
704:     smart_str buf = 0;
705:
706:     if (zend_parse_parameters_none() == FAILURE) {
707:         return;
708:     }
709:
710:     PHP_VAR_SERIALIZE_INIT(var_hash);
711:
712:     /* storage */
713:     smart_str_append(&buf, "s:", 2);
714:     ZVAL_LONG(&flags, zend_hash_num_elements(intern->storage));
715:     PHP_VAR_SERIALIZE(&buf, &flags, &var_hash);
716:     smart_str_append(&buf, "f:", 2);
717:
718:     zend_hash_internal_pointer_reset_ex(intern->storage, &pos);
719:
720:     while ((element = zend_hash_get_current_data_ptr_ex(intern->storage, &pos)) != NULL) {
721:         if (element = zend_hash_get_current_data_ptr_ex(intern->storage, &pos) == NULL) {
722:             smart_str_free(&buf);
723:             RETURN_NULL();
724:         }
725:         smart_str_append(&buf, element->obj, &var_hash);
726:         smart_str_append(&buf, ":", 1);
727:         smart_str_append(&buf, element->inf, &var_hash);
728:         smart_str_append(&buf, ":", 1);
729:         zend_hash_move_forward_ex(intern->storage, &pos);
730:     }
731:
732:     /* members */
733:     smart_str_append(&buf, "m:", 2);
734:
735:     ZVAL_ARR(&members, zend_array_dup(zend_std_get_grogeties(getThis())));
736:     PHP_VAR_SERIALIZE(&buf, &members, &var_hash); /* finishes the string */
737:     smart_str_free(&members);
738:
739:     /* done */
740:     PHP_VAR_SERIALIZE_DESTROY(var_hash);
741:
742:     if (buf.s) {
743:         RETURN_NEW_STR(buf.s);
744:     } else {
745:         RETURN_NULL();
746:     }
747: } /* }}} */
748:
749: /* {{{ proto void SplObjectStorage::unserialize(string $serialized)
750:  Unserializes storage */

```

```

753: SPL_METHOD(splObjectStorage, unserialize)
754: {
755:     splObjectStorage *intern = Z_SPLOBJ_STORAGE_P(getThis());
756:
757:     char *buf;
758:     size_t buf_len;
759:     const unsigned char *p;
760:     php_unserialize_data_t var_hash;
761:     zval entry, info;
762:     zval *pcount, *pmembers;
763:     splObjectStorageElement *element;
764:     zend_long count;
765:
766:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &buf, &buf_len) == FAILURE) {
767:         return;
768:     }
769:
770:     if (buf_len == 0) {
771:         return;
772:     }
773:
774:     /* storage */
775:     s = p = (const unsigned char*)buf;
776:     PHP_VAR_UNSERIALIZE_INIT(var_hash);
777:
778:     if (*p != 'p' || **p != ':' || *++p != ':') {
779:         goto outexcept;
780:     }
781:     **p;
782:
783:     pcount = var_tmp_var(&var_hash);
784:     if (fphp_var_unserialize(pcount, &p, s + buf_len, &var_hash) != IS_LONG) {
785:         goto outexcept;
786:     }
787:
788:     --p; /* for 'p' */
789:     count = Z_VAL_P(pcount);
790:
791:     ZVAL_UNDEF(&entry);
792:     ZVAL_UNDEF(&info);
793:
794:     while (count-- > 0) {
795:         splObjectStorageElement *element;
796:         zend_hash_key key;
797:
798:         if (*p != ':' || *++p != ':') {
799:             goto outexcept;
800:         }
801:         **p;
802:         if (*p != 'O' || **p != 'C' || *++p != 'C' || *++p != 'r') {
803:             goto outexcept;
804:         }
805:         /* store reference to allow cross-references between different elements */
806:         if (fphp_var_unserialize(&entry, &p, s + buf_len, &var_hash) {
807:             goto outexcept;
808:         }
809:         if (*p == ',' || /* new version has inf */)
810:             **p;
811:         if (fphp_var_unserialize(&info, &p, s + buf_len, &var_hash) {
812:             goto outexcept;
813:         }
814:
815:         if (Z_TYPE(entry) != IS_OBJECT) {
816:             zval_ptr_dtor(&entry);
817:             zval_ptr_dtor(&info);
818:             goto outexcept;
819:         }
820:
821:         if (spl_object_storage_get_hash(&key, intern, getThis(), &entry) == FAILURE) {
822:             zval_ptr_dtor(&entry);
823:             zval_ptr_dtor(&info);
824:             goto outexcept;
825:         }
826:
827:         element = spl_object_storage_get(intern, &key);
828:         spl_object_storage_free_hash(intern, &key);
829:         if (element) {
830:             if (!Z_UNDEF(element->info)) {
831:                 var_push_dtor(&var_hash, &element->info);
832:             }
833:             if (!Z_UNDEF(element->obj)) {
834:                 var_push_dtor(&var_hash, &element->obj);
835:             }
836:         }
837:         element = spl_object_storage_attach(intern, getThis(), &entry, Z_UNDEF(&info)?NULL:&info);
838:         var_replace(&var_hash, &entry, &element->obj);
839:         var_replace(&var_hash, &info, &element->info);
840:         zval_ptr_dtor(&entry);
841:         ZVAL_UNDEF(&entry);
842:         zval_ptr_dtor(&info);
843:         ZVAL_UNDEF(&info);
844:     }
845:
846:     if (*p != ':' || *++p != ':') {
847:         goto outexcept;
848:     }
849:     **p;
850:
851:     /* members */
852:     if (*p != 'm' || **p != ':' || *++p != ':') {
853:         goto outexcept;
854:     }
855:     **p;
856:
857:     pmembers = var_tmp_var(&var_hash);
858:     if (fphp_var_unserialize(pmembers, &p, s + buf_len, &var_hash) || Z_TYPE_P(pmembers) != IS_ARRAY) {
859:         goto outexcept;
860:     }
861:
862:     /* copy members */
863:     object_properties_load(&intern->std, Z_ARRVAL_P(pmembers));
864:
865:     PHP_VAR_UNSERIALIZE_DESTROY(&var_hash);
866:     return;
867:
868: outexcept:
869:     PHP_VAR_UNSERIALIZE_DESTROY(&var_hash);
870:     zend_throw_exception_ex(spl_ce_UnexpectedValueException, 0, "Error at offset %d of %d bytes", ((char*)p - buf), buf_len);
871:     return;
872:
873:     /* }}} */
874:
875: ZEND_BEGIN_ARG_INFO(arginfo_Object, 0)
876:     ZEND_ARG_INFO(0, object)
877: ZEND_END_ARG_INFO()
878:
879: ZEND_BEGIN_ARG_INFO_EX(arginfo_Attach, 0, 0, 1)
880:     ZEND_ARG_INFO(0, object)
881:     ZEND_ARG_INFO(0, info)
882: ZEND_END_ARG_INFO()
883:
884: ZEND_BEGIN_ARG_INFO(arginfo_Serialize, 0)
885:     ZEND_ARG_INFO(0, serialized)
886: ZEND_END_ARG_INFO()
887:
888: ZEND_BEGIN_ARG_INFO(arginfo_GetInfo, 0)
889:     ZEND_ARG_INFO(0, info)
890: ZEND_END_ARG_INFO()
891:
892: ZEND_BEGIN_ARG_INFO(arginfo_GetHash, 0)
893:     ZEND_ARG_INFO(0, object)
894: ZEND_END_ARG_INFO()
895:
896: ZEND_BEGIN_ARG_INFO_EX(arginfo_OffsetGet, 0, 0, 1)
897:     ZEND_ARG_INFO(0, object)
898: ZEND_END_ARG_INFO()
899:
900: ZEND_BEGIN_ARG_INFO(arginfo_splobjject_void, 0)
901: ZEND_END_ARG_INFO()
902:
903: static const zend_function_entry spl_funcs_splObjectStorage[] = {
904:     SPL_ME(splObjectStorage, attach, arginfo_attach, 0),
905:     SPL_ME(splObjectStorage, detach, arginfo_detach, 0),
906:     SPL_ME(splObjectStorage, contains, arginfo_contains, 0),
907:     SPL_ME(splObjectStorage, addAll, arginfo_addAll, 0),
908:     SPL_ME(splObjectStorage, removeAll, arginfo_removeAll, 0),
909:     SPL_ME(splObjectStorage, removeAllExcept, arginfo_removeAllExcept, 0),
910:     SPL_ME(splObjectStorage, getInfo, arginfo_splobjject_void, 0),
911:     SPL_ME(splObjectStorage, setInfo, arginfo_setInfo, 0),
912:     SPL_ME(splObjectStorage, getHash, arginfo_getHash, 0),
913:     /* Constant */
914:     SPL_ME(splObjectStorage, count, arginfo_splobjject_void, 0),
915:     /* Iterator */
916:     SPL_ME(splObjectStorage, rewind, arginfo_splobjject_void, 0),
917:     SPL_ME(splObjectStorage, valid, arginfo_splobjject_void, 0),
918:     SPL_ME(splObjectStorage, key, arginfo_splobjject_void, 0),
919:     SPL_ME(splObjectStorage, current, arginfo_splobjject_void, 0),
920:     SPL_ME(splObjectStorage, next, arginfo_splobjject_void, 0),
921:     /* Serializable */
922:     SPL_ME(splObjectStorage, unserialize, arginfo_Serialize, 0),
923:     SPL_ME(splObjectStorage, serialize, arginfo_splobjject_void, 0),
924:     /* ArrayAccess */
925:     SPL_MA(splObjectStorage, offsetExists, splObjectStorage, contains, arginfo_OffsetGet, 0),
926:     SPL_MA(splObjectStorage, offsetSet, splObjectStorage, attach, arginfo_attach, 0),
927:     SPL_MA(splObjectStorage, offsetUnset, splObjectStorage, detach, arginfo_OffsetGet, 0),
928:     SPL_ME(splObjectStorage, offsetGet, arginfo_OffsetGet, 0),
929:     PHP_FE_END
930: };
931:
932: typedef enum {
933:     MIT_NEED_ANY = 0,
934:     MIT_NEED_ALL = 1,
935:     MIT_KEYS_NUMERIC = 0,
936:     MIT_KEYS_ASSOC = 2
937: } MultiplierIteratorFlags;
938:
939: #define SPL_MULTIPLE_ITERATOR_GET_ALL_CW 1
940: #define SPL_MULTIPLE_ITERATOR_GET_ALL_KEY 2

```

```

941:
942: /* {{{ proto void MultiplierIterator::__construct([int flags = MIT_NEED_ALL|MIT_KEYS_NUMERIC])
943:    Iterator that iterates over several iterators one after the other */
944: SPL_METHOD(MultiplierIterator, __construct)
945: {
946:     splObjectStorage *intern;
947:     zend_long flags = MIT_NEED_ALL|MIT_KEYS_NUMERIC;
948:
949:     if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "i", &flags) == FAILURE) {
950:         return;
951:     }
952:
953:     intern = Z_SPLOBJ_STORAGE_P(getThis());
954:     intern->flags = flags;
955: }
956: /* }}} */
957:
958: /* {{{ proto int MultiplierIterator::getFlags()
959:    Return current flags */
960: SPL_METHOD(MultiplierIterator, getFlags)
961: {
962:     splObjectStorage *intern = Z_SPLOBJ_STORAGE_P(getThis());
963:
964:     if (zend_parse_parameters_none() == FAILURE) {
965:         return;
966:     }
967:     RETURN_LONG(intern->flags);
968: }
969: /* }}} */
970:
971: /* {{{ proto int MultiplierIterator::setFlags(int flags)
972:    Set flags */
973: SPL_METHOD(MultiplierIterator, setFlags)
974: {
975:     splObjectStorage *intern;
976:     intern = Z_SPLOBJ_STORAGE_P(getThis());
977:
978:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "i", &intern->flags) == FAILURE) {
979:         return;
980:     }
981: }
982: /* }}} */
983:
984: /* {{{ proto void attachIterator(Iterator iterator[, mixed info]) throws InvalidArgumentException
985:    Attach a new iterator */
986: SPL_METHOD(MultiplierIterator, attachIterator)
987: {
988:     splObjectStorage *intern;
989:     zval *iterator = NULL, *info = NULL;
990:
991:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "Ois", &iterator, &zend_obj_iterator, &info) == FAILURE) {
992:         return;
993:     }
994:
995:     intern = Z_SPLOBJ_STORAGE_P(getThis());
996:
997:     if (info != NULL) {
998:         splObjectStorageElement *element;
999:
1000:         if (Z_TYPE_P(info) != IS_LONG || Z_TYPE_P(info) != IS_STRING) {
1001:             zend_throw_exception(spl_ce_InvalidArgumentException, "Info must be NULL, integer or string", 0);
1002:             return;
1003:         }
1004:
1005:         zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1006:         while (element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos) != NULL) {
1007:             if (fast_is_identical_function(info, element->info)) {
1008:                 zend_throw_exception(spl_ce_InvalidArgumentException, "Key duplication error", 0);
1009:                 return;
1010:             }
1011:             zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1012:         }
1013:
1014:         spl_objject_storage_attach(intern, getThis(), iterator, info);
1015:     }
1016: }
1017: /* }}} */
1018:
1019: /* {{{ proto void MultiplierIterator::rewind()
1020:    Rewind all attached iterator instances */
1021: SPL_METHOD(MultiplierIterator, rewind)
1022: {
1023:     splObjectStorage *intern;
1024:     splObjectStorageElement *element;
1025:     zval *it;
1026:
1027:     intern = Z_SPLOBJ_STORAGE_P(getThis());
1028:
1029:     if (zend_parse_parameters_none() == FAILURE) {
1030:         return;
1031:     }
1032:
1033:     zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1034:     while (element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos) != NULL || IS(exception)) {
1035:         if (element->obj) {
1036:             zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs, ZEND_FUNC_REWIND, NULL);
1037:             zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1038:         }
1039:     }
1040: }
1041: /* }}} */
1042:
1043: /* {{{ proto void MultiplierIterator::next()
1044:    Move all attached iterator instances forward */
1045: SPL_METHOD(MultiplierIterator, next)
1046: {
1047:     splObjectStorage *intern;
1048:     splObjectStorageElement *element;
1049:     zval *it;
1050:
1051:     intern = Z_SPLOBJ_STORAGE_P(getThis());
1052:
1053:     if (zend_parse_parameters_none() == FAILURE) {
1054:         return;
1055:     }
1056:
1057:     zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1058:     while (element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos) != NULL || IS(exception)) {
1059:         if (element->obj) {
1060:             zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs, ZEND_FUNC_NEXT, NULL);
1061:             zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1062:         }
1063:     }
1064: }
1065: /* }}} */
1066:
1067: /* {{{ proto bool MultiplierIterator::valid()
1068:    Return whether all or one sub iterator is valid depending on flags */
1069: SPL_METHOD(MultiplierIterator, valid)
1070: {
1071:     splObjectStorage *intern;
1072:     zval *it, *retval;
1073:     zend_long expect, valid;
1074:
1075:     intern = Z_SPLOBJ_STORAGE_P(getThis());
1076:
1077:     if (zend_parse_parameters_none() == FAILURE) {
1078:         return;
1079:     }
1080:
1081:     if (zend_hash_num_elements(&intern->storage) {
1082:         RETURN_FALSE;
1083:     }
1084:
1085:     expect = (intern->flags & MIT_NEED_ALL) ? 1 : 0;
1086:
1087:     zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1088:     while (element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos) != NULL || IS(exception)) {
1089:         if (element->obj) {
1090:             zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs, ZEND_FUNC_VALID, &retval);
1091:         }
1092:         if (!Z_UNDEF(retval)) {
1093:             valid = (Z_TYPE(retval) == IS_TRUE);
1094:             zval_ptr_dtor(&retval);
1095:         } else {
1096:             valid = 0;
1097:         }
1098:     }
1099:     if (expect != valid) {
1100:         RETURN_BOOL(!expect);
1101:     }
1102: }
1103:
1104: zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1105:
1106: RETURN_BOOL(expect);
1107: }
1108: /* }}} */
1109:
1110: static void spl_multiple_iterator_get_all(splObjectStorage *intern, int get_type, zval *return_value) /* {{{ */
1111: {
1112:     splObjectStorageElement *element;
1113:     zval *it, *retval;
1114:     int valid = 1, num_elements;
1115:
1116:     num_elements = zend_hash_num_elements(&intern->storage);
1117:     if (num_elements < 1) {
1118:         RETURN_FALSE;
1119:     }
1120:
1121:     array_init_size(return_value, num_elements);
1122:
1123:     zend_hash_internal_pointer_reset_ex(&intern->storage, &intern->pos);
1124:     while (element = zend_hash_get_current_data_ptr_ex(&intern->storage, &intern->pos) != NULL || IS(exception)) {
1125:         if (element->obj) {
1126:             zend_call_method_with_0_params(it, Z_OBJCE_P(it), &Z_OBJCE_P(it)->iterator_funcs, ZEND_FUNC_VALID, &retval);
1127:         }
1128:         if (!Z_UNDEF(retval)) {
1129:             valid = (Z_TYPE(retval) == IS_TRUE);

```



```

1129:     rval_get_dtor(&retval);
1130: } else {
1131:     valid = 0;
1132: }
1133:
1134: if (valid) {
1135:     if (SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT == get_type) {
1136:         send_call_method_with_0_params(it, z_OBJCE_P(it), z_OBJCE_P(it) -> iterator_funcs.if_current, "current", &retval);
1137:     } else {
1138:         send_call_method_with_0_params(it, z_OBJCE_P(it), z_OBJCE_P(it) -> iterator_funcs.if_key, "key", &retval);
1139:     }
1140:     if (!Z_ISUNDEF(retval)) {
1141:         zend_throw_exception(spl_ce_RuntimeException, "Failed to call sub iterator method", 0);
1142:         return;
1143:     }
1144: } else if (intern->flags & MIT_NEED_ALL) {
1145:     if (SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT == get_type) {
1146:         zend_throw_exception(spl_ce_RuntimeException, "Called current() with non valid sub iterator", 0);
1147:     } else {
1148:         zend_throw_exception(spl_ce_RuntimeException, "Called key() with non valid sub iterator", 0);
1149:     }
1150:     return;
1151: } else {
1152:     ZVAL_NULL(&retval);
1153: }
1154:
1155: if (intern->flags & MIT_KEYS_ASSOC) {
1156:     switch (z_TYPE(element->info)) {
1157:         case IS_LONG:
1158:             add_index_zval(return_value, z_IVAL(element->info), &retval);
1159:             break;
1160:         case IS_STRING:
1161:             zend_symtable_update(z_ARRVAL_P(return_value), z_STR(element->info), &retval);
1162:             break;
1163:         default:
1164:             rval_get_dtor(&retval);
1165:             zend_throw_exception(spl_ce_InvalidArgumentException, "Sub-Iterator is associated with NULL", 0);
1166:             return;
1167:     }
1168: } else {
1169:     add_next_index_zval(return_value, &retval);
1170: }
1171:
1172: zend_hash_move_forward_ex(&intern->storage, &intern->pos);
1173: }
1174: /* }}} */
1175: /* }}} */
1176:
1177: /* {{{ proto array current() throws RuntimeException throws InvalidArgumentException
1178:    Return an array of all registered Iterator instances current() result */
1179: SPL_METHOD(MultipleIterator, current)
1180: {
1181:     spl_SplObjectStorage *intern;
1182:     intern = z_SPL_OBJECT_STORAGE_P(getThis());
1183:
1184:     if (zend_parse_parameters_none() == FAILURE) {
1185:         return;
1186:     }
1187:
1188:     spl_multiple_iterator_get_all(intern, SPL_MULTIPLE_ITERATOR_GET_ALL_CURRENT, return_value);
1189: }
1190: /* }}} */
1191:
1192: /* {{{ proto array MultipleIterator::key()
1193:    Return an array of all registered Iterator instances key() result */
1194: SPL_METHOD(MultipleIterator, key)
1195: {
1196:     spl_SplObjectStorage *intern;
1197:     intern = z_SPL_OBJECT_STORAGE_P(getThis());
1198:
1199:     if (zend_parse_parameters_none() == FAILURE) {
1200:         return;
1201:     }
1202:
1203:     spl_multiple_iterator_get_all(intern, SPL_MULTIPLE_ITERATOR_GET_ALL_KEY, return_value);
1204: }
1205: /* }}} */
1206:
1207: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_attachIterator, 0, 0, 1)
1208:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
1209: ZEND_END_ARG_INFO();
1210: ZEND_FUNC_INFO();
1211:
1212: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_detachIterator, 0, 0, 1)
1213:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
1214: ZEND_END_ARG_INFO();
1215:
1216: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_containsIterator, 0, 0, 1)
1217:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
1218: ZEND_END_ARG_INFO();
1219:
1220: ZEND_BEGIN_ARG_INFO_EX(arginfo_MultipleIterator_setFlags, 0, 0, 1)
1221:     ZEND_ARG_INFO(0, flags)
1222: ZEND_END_ARG_INFO();
1223:
1224: static const zend_function_entry spl_funcs_MultipleIterator[] = {
1225:     SPL_ME(MultipleIterator, __construct,      arginfo_MultipleIterator_setFlags,      0)
1226:     SPL_ME(MultipleIterator, getFlags,         arginfo_splObject_void,                0)
1227:     SPL_ME(MultipleIterator, setFlags,         arginfo_MultipleIterator_setFlags,     0)
1228:     SPL_ME(MultipleIterator, attachIterator,   arginfo_MultipleIterator_attachIterator, 0)
1229:     SPL_MA(MultipleIterator, detachIterator,   splObjectStorage, detach,      arginfo_MultipleIterator_detachIterator, 0)
1230:     SPL_MA(MultipleIterator, containsIterator, splObjectStorage, contains,   arginfo_MultipleIterator_containsIterator, 0)
1231:     SPL_MA(MultipleIterator, countIterators,  splObjectStorage, count,     arginfo_splObject_void,          0)
1232:     /* Iterator */
1233:     SPL_ME(MultipleIterator, rewind,           arginfo_splObject_void,            0)
1234:     SPL_ME(MultipleIterator, valid,           arginfo_splObject_void,            0)
1235:     SPL_ME(MultipleIterator, key,             arginfo_splObject_void,            0)
1236:     SPL_ME(MultipleIterator, current,         arginfo_splObject_void,            0)
1237:     SPL_ME(MultipleIterator, next,            arginfo_splObject_void,            0)
1238:     PHP_FE_END
1239: };
1240:
1241: /* {{{ PHP_MINIT_FUNCTION(spl_observer) */
1242: PHP_MINIT_FUNCTION(spl_observer)
1243: {
1244:     REGISTER_SPL_INTERFACE(splObserver);
1245:     REGISTER_SPL_INTERFACE(splSubject);
1246:
1247:     REGISTER_SPL_STD_CLASS_EX(splObjectStorage, spl_SplObjectStorage_new, spl_funcs_splObjectStorage);
1248:     memcpy(&spl_handler_splObjectStorage, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1249:
1250:     spl_handler_splObjectStorage.offset = XOffsetOf(spl_SplObjectStorage, std);
1251:     spl_handler_splObjectStorage.get_debug_info = spl_object_storage_debug_info;
1252:     spl_handler_splObjectStorage.compare_objects = spl_object_storage_compare_objects;
1253:     spl_handler_splObjectStorage.clone_obj = spl_object_storage_clone;
1254:     spl_handler_splObjectStorage.get_gc = spl_object_storage_get_gc;
1255:     spl_handler_splObjectStorage.dtor_obj = zend_object_dtor_obj;
1256:     spl_handler_splObjectStorage.free_obj = spl_SplObjectStorage_free_storage;
1257:
1258:     REGISTER_SPL_IMPLEMENTATIONS(splObjectStorage, Countable);
1259:     REGISTER_SPL_IMPLEMENTATIONS(splObjectStorage, Iterator);
1260:     REGISTER_SPL_IMPLEMENTATIONS(splObjectStorage, Serializable);
1261:     REGISTER_SPL_IMPLEMENTATIONS(splObjectStorage, ArrayAccess);
1262:
1263:     REGISTER_SPL_STD_CLASS_EX(MultipleIterator, spl_SplObjectStorage_new, spl_funcs_MultipleIterator);
1264:     REGISTER_SPL_ITERATOR(MultipleIterator);
1265:
1266:     REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_NEED_ANY", MIT_NEED_ANY);
1267:     REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_NEED_ALL", MIT_NEED_ALL);
1268:     REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_KEYS_NUMERIC", MIT_KEYS_NUMERIC);
1269:     REGISTER_SPL_CLASS_CONST_LONG(MultipleIterator, "MIT_KEYS_ASSOC", MIT_KEYS_ASSOC);
1270:
1271:     return SUCCESS;
1272: }
1273: /* }}} */
1274:
1275: /*
1276:  * Local variables:
1277:  * tab-width: 4
1278:  * c-basic-offset: 4
1279:  * End:
1280:  * vim600: fdm=marker
1281:  * vim: noet sw=4 ts=4
1282:  */

```

```

1: 1. PHP Version 7
2: |
3: | Copyright (c) 1997-2018 The PHP Group
4: |
5: | This source file is subject to version 3.01 of the PHP license,
6: | that is bundled with this package in the file LICENSE, and is
7: | available through the world-wide-web at the following url:
8: | http://www.php.net/licenses/3.01.txt
9: |
10: | If you did not receive a copy of the PHP license and are unable to
11: | obtain it through the world-wide-web, please send a note to
12: | license@php.net so we can mail you a copy immediately.
13: |
14: | Authors: Stigme Kneuss <olderphp@net>
15: |
16: |
17: */
18:
19: /* $Id$ */
20:
21: #ifdef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "php.h"
26: #include "zend_exceptions.h"
27: #include "zend_hash.h"
28:
29: #include "php_spl.h"
30: #include "ext/standard/info.h"
31: #include "ext/standard/php_var.h"
32: #include "zend_smart_str.h"
33: #include "spl_functions.h"
34: #include "spl_engine.h"
35: #include "spl_iterators.h"
36: #include "spl_dlist.h"
37: #include "spl_exceptions.h"
38:
39: zend_object_handlers spl_handler_spl doubly linked list;
40: PHP_API zend_class_entry *spl_cs_spl doubly linked list;
41: PHP_API zend_class_entry *spl_cs_spl doubly linked list;
42: PHP_API zend_class_entry *spl_cs_spl doubly linked list;
43:
44: #define SPL_LIST_DELETE(elem) if (!--(elem->ret) && !\
45: elem->data) {
46: }
47:
48: #define SPL_LIST_CHECK_DELETE(elem) if (elem) && !--(elem->ret) && !\
49: elem->data) {
50: }
51:
52: #define SPL_LIST_ADDREF(elem) (elem)->ret++
53: #define SPL_LIST_CHECK_ADDREF(elem) if (elem) (elem)->ret++
54:
55: #define SPL_LIST_IT_DELETE 0x00000001 /* Delete flag makes the iterator delete the current element on next */
56: #define SPL_LIST_IT_MASK 0x00000003 /* LIFO flag makes the iterator traverse the structure as a last-in-first-out */
57: #define SPL_LIST_IT_FIX 0x00000004 /* Backward/Forward bit is fixed */
58:
59: #ifdef HAVE_ACCEPT
60: #define accept
61: #endif
62:
63:
64: typedef struct _spl_ptr_list_element {
65: struct _spl_ptr_list_element *prev;
66: struct _spl_ptr_list_element *next;
67: int ret;
68: void *data;
69: } spl_ptr_list_element;
70:
71: typedef void (*spl_ptr_list_dtor_func)(spl_ptr_list_element *);
72: typedef void (*spl_ptr_list_ctor_func)(spl_ptr_list_element *);
73:
74: typedef struct _spl_ptr_list {
75: spl_ptr_list_element *head;
76: spl_ptr_list_element *tail;
77: spl_ptr_list_dtor_func dtor;
78: spl_ptr_list_ctor_func ctor;
79: int count;
80: } spl_ptr_list;
81:
82: typedef struct _spl_dlist_object spl_dlist_object;
83: typedef struct _spl_dlist_it spl_dlist_it;
84:
85: struct _spl_dlist_object {
86: spl_ptr_list *list;
87: int traverse_position;
88: spl_ptr_list_element *traverse_pointer;
89: int flags;
90: zend_function *fptr_offset_get;
91: zend_function *fptr_offset_set;
92: zend_function *fptr_offset_hack;
93: zend_function *fptr_offset_del;
94: zend_function *fptr_count;
95: zend_class_entry *ce_get_iterator;
96: void *gc_data;
97: int gc_data_count;
98: zend_object std;
99: };
100:
101: /* Define an overloaded iterator structure */
102: struct _spl_dlist_it {
103: zend_weak_iterator intern;
104: spl_ptr_list_element *traverse_pointer;
105: int traverse_position;
106: int flags;
107: };
108:
109: static inline spl_dlist_object *spl_dlist_from_obj(zend_object *obj) { /* {{{ */
110: return (spl_dlist_object*)((char*)(obj) - XtOffsetOf(spl_dlist_object, std));
111: }
112: /* }}} */
113:
114: #define SPL_DLIST_P(obj) spl_dlist_from_obj(&obj->std)
115:
116: /* {{{ spl_ptr_list */
117: static void spl_ptr_list_val_ctor(spl_ptr_list_element *elem) { /* {{{ */
118: if (!IS_ITERATOR(elem->data)) {
119: elem->data = (void*)0;
120: ZVAL_UNDEF(elem->data);
121: }
122: }
123: /* }}} */
124:
125: static void spl_ptr_list_val_dtor(spl_ptr_list_element *elem) { /* {{{ */
126: if (!IS_ITERATOR(elem->data)) {
127: Z_ADDREF(elem->data);
128: }
129: }
130: /* }}} */
131:
132: static spl_ptr_list *spl_ptr_list_init(spl_ptr_list_ctor_func ctor, spl_ptr_list_dtor_func dtor) { /* {{{ */
133: {
134: spl_ptr_list *list = emalloc(sizeof(spl_ptr_list));
135:
136: list->head = NULL;
137: list->tail = NULL;
138: list->count = 0;
139: list->dtor = dtor;
140: list->ctor = ctor;
141:
142: return list;
143: }
144: /* }}} */
145:
146: static zend_long spl_ptr_list_count(spl_ptr_list *list) { /* {{{ */
147: return (zend_long)list->count;
148: }
149:
150: /* }}} */
151:
152: static void spl_ptr_list_destroy(spl_ptr_list *list) { /* {{{ */
153: {
154: spl_ptr_list_element *current = list->head, *next;
155: spl_ptr_list_dtor_func dtor = list->dtor;
156:
157: while (current) {
158: next = current->next;
159: if (dtor) {
160: dtor(current);
161: }
162: SPL_LIST_DELETE(current);
163: current = next;
164: }
165:
166: efree(list);
167: }
168: /* }}} */
169:
170: static spl_ptr_list_element *spl_ptr_list_offset(spl_ptr_list *list, zend_long offset, int backward) { /* {{{ */
171: {
172: spl_ptr_list_element *current;
173: int pos = 0;
174:
175: if (backward) {
176: current = list->tail;
177: } else {
178: current = list->head;
179: }
180:
181: while (current && pos < offset) {
182: pos++;
183: if (backward) {
184: current = current->prev;
185: } else {
186: current = current->next;
187: }
188: }
189:
190: return current;
191: }
192:
193: static void spl_ptr_list_push(spl_ptr_list *list, void *data) { /* {{{ */
194: {
195: spl_ptr_list_element *elem = emalloc(sizeof(spl_ptr_list_element));
196:
197: elem->prev = NULL;
198: elem->next = list->head;
199: ZVAL_COPY_VALUE(elem->data, data);
200:
201: if (list->head) {
202: list->head->prev = elem;
203: } else {
204: list->tail = elem;
205: }
206:
207: list->head = elem;
208: list->count++;
209: }
210:
211: static void spl_ptr_list_pop(spl_ptr_list *list, void *ret) { /* {{{ */
212: {
213: spl_ptr_list_element *elem = emalloc(sizeof(spl_ptr_list_element));
214:
215: elem->prev = NULL;
216: elem->next = list->tail;
217: ZVAL_COPY_VALUE(elem->data, data);
218:
219: if (list->tail) {
220: list->tail->next = elem;
221: } else {
222: list->head = elem;
223: }
224:
225: list->tail = elem;
226: list->count++;
227:
228: if (list->ret) {
229: list->ret = elem->data;
230: } else {
231: list->ret = elem;
232: }
233:
234: list->tail = elem;
235: list->count++;
236:
237: if (list->ret) {
238: list->ret = elem;
239: }
240:
241: /* }}} */
242:
243: static void spl_ptr_list_pop(spl_ptr_list *list, void *ret) { /* {{{ */
244: {
245: spl_ptr_list_element *tail = list->tail;
246:
247: if (tail == NULL) {
248: ZVAL_UNDEF(ret);
249: return;
250: }
251:
252: if (tail->prev) {
253: tail->prev->next = NULL;
254: } else {
255: list->head = NULL;
256: }
257:
258: list->tail = tail->prev;
259: list->count--;
260: ZVAL_COPY(ret, tail->data);
261:
262: if (list->dtor) {
263: list->dtor(tail);
264: }
265:
266: ZVAL_UNDEF(tail->data);
267:
268: SPL_LIST_DELETE(tail);
269: }
270: /* }}} */
271:
272: static void spl_ptr_list_last(spl_ptr_list *list) { /* {{{ */
273: {
274: spl_ptr_list_element *tail = list->tail;
275:
276: if (tail == NULL) {
277: return NULL;
278: } else {
279: return tail->data;
280: }
281: }
282: /* }}} */
283:
284: static void spl_ptr_list_first(spl_ptr_list *list) { /* {{{ */
285: {
286: spl_ptr_list_element *head = list->head;
287:
288: if (head == NULL) {
289: return NULL;
290: } else {
291: return head->data;
292: }
293: }
294: /* }}} */
295:
296: static void spl_ptr_list_shift(spl_ptr_list *list, void *ret) { /* {{{ */
297: {
298: spl_ptr_list_element *head = list->head;
299:
300: if (head == NULL) {
301: ZVAL_UNDEF(ret);
302: return;
303: }
304:
305: if (head->next) {
306: head->next->prev = NULL;
307: } else {
308: list->tail = NULL;
309: }
310:
311: list->head = head->next;
312: list->count--;
313: ZVAL_COPY(ret, head->data);
314:
315: if (list->dtor) {
316: list->dtor(head);
317: }
318: ZVAL_UNDEF(head->data);
319:
320: SPL_LIST_DELETE(head);
321: }
322: /* }}} */
323:
324: static void spl_ptr_list_copy(spl_ptr_list *from, spl_ptr_list *to) { /* {{{ */
325: {
326: spl_ptr_list_element *current = from->head, *next;
327: /* ??? spl_ptr_list_ctor_func ctor = from->ctor; */
328:
329: while (current) {
330: next = current->next;
331:
332: /* ??? FIXME */
333: if (ctor) {
334: ctor(current);
335: }
336:
337: spl_ptr_list_push(to, current->data);
338: current = next;
339: }
340:
341: }
342:
343: static void spl_dlist_object_free_storage(zend_object *obj) { /* {{{ */
344: {
345: spl_dlist_object *intern = spl_dlist_from_obj(obj);
346: void *tmp;
347:
348: while (intern->list->count > 0) {
349: spl_ptr_list_pop(spl_ptr_list *list, tmp);
350: ZVAL_COPY(tmp, tmp);
351: }
352:
353: if (intern->gc_data == NULL) {
354: efree(intern->gc_data);
355: }
356:
357: spl_ptr_list_destroy(intern->list);
358: SPL_LIST_CHECK_DELETE(intern->traverse_pointer);
359: }
360: /* }}} */
361:
362: static zend_object_iterator *spl_dlist_get_iterator(zend_class_entry *ce, void *obj, zend_object_iterator *parent, zend_object_iterator *class_type, void *data) {
363: {
364: spl_dlist_object *intern = spl_dlist_from_obj(obj);
365: int pos = 0;
366:
367: if (parent) {
368: return parent;
369: }
370:
371: return zend_object_iterator_new(spl_dlist_object, parent, pos);
372: }
373:
374: static
```

```

377: zend_object_std_init(&intern->std, class_type);
378: object_properties_init(&intern->std, class_type);
379:
380:
381: intern->flags = 0;
382: intern->traverse_position = 0;
383:
384: if (orig) {
385:     spl_dlist_object *other = Z_SPDOLLIST_P(orig);
386:     intern->oc_get_iterator = other->oc_get_iterator;
387:
388:     if (clone_orig) {
389:         intern->llist = (spl_ptr_llist *)spl_ptr_llist_init(&other->llist->ctor, &other->llist->dtor);
390:         spl_ptr_llist_copy(&other->llist, intern->llist);
391:         intern->traverse_pointer = intern->llist->thead;
392:         SPL_LLIST_CHECK_ADDRP(intern->traverse_pointer);
393:     } else {
394:         intern->llist = other->llist;
395:         intern->traverse_pointer = intern->llist->thead;
396:         SPL_LLIST_CHECK_ADDRP(intern->traverse_pointer);
397:     }
398:
399:     intern->flags = other->flags;
400: } else {
401:     intern->llist = (spl_ptr_llist *)spl_ptr_llist_init(spl_ptr_llist_ctor_val, spl_ptr_llist_dtor);
402:     intern->traverse_pointer = intern->llist->thead;
403:     SPL_LLIST_CHECK_ADDRP(intern->traverse_pointer);
404: }
405:
406: while (parent) {
407:     if (parent == spl_oc_SplStack) {
408:         intern->flags |= (SPL_DLIST_IT_FIX | SPL_DLIST_IT_IFPO);
409:         intern->std.handlers = spl_handler_SplDoublyLinkedList;
410:     } else if (parent == spl_oc_SplQueue) {
411:         intern->flags |= (SPL_DLIST_IT_FIX);
412:         intern->std.handlers = spl_handler_SplDoublyLinkedList;
413:     }
414:
415:     if (parent == spl_oc_SplDoublyLinkedList) {
416:         intern->std.handlers = spl_handler_SplDoublyLinkedList;
417:         break;
418:     }
419:
420:     parent = parent->parent;
421:     inherited = 1;
422: }
423:
424: if (!parent) { /* this must never happen */
425:     php_error_docref(NULL, E_COMPILE_ERROR, "Internal compiler error, Class is not child of SplDoublyLinkedList");
426: }
427:
428: if (inherited) {
429:     intern->fptr_offset_get = zend_hash_str_find_ptr(class_type->function_table, "offsetget", sizeof("offsetget") - 1);
430:     if (intern->fptr_offset_get->common.scope == parent) {
431:         intern->fptr_offset_get = NULL;
432:     }
433:     intern->fptr_offset_set = zend_hash_str_find_ptr(class_type->function_table, "offsetset", sizeof("offsetset") - 1);
434:     if (intern->fptr_offset_set->common.scope == parent) {
435:         intern->fptr_offset_set = NULL;
436:     }
437:     intern->fptr_offset_has = zend_hash_str_find_ptr(class_type->function_table, "offsetexists", sizeof("offsetexists") - 1);
438:     if (intern->fptr_offset_has->common.scope == parent) {
439:         intern->fptr_offset_has = NULL;
440:     }
441:     intern->fptr_offset_del = zend_hash_str_find_ptr(class_type->function_table, "offsetunset", sizeof("offsetunset") - 1);
442:     if (intern->fptr_offset_del->common.scope == parent) {
443:         intern->fptr_offset_del = NULL;
444:     }
445:     intern->fptr_count = zend_hash_str_find_ptr(class_type->function_table, "count", sizeof("count") - 1);
446:     if (intern->fptr_count->common.scope == parent) {
447:         intern->fptr_count = NULL;
448:     }
449: }
450:
451: return &intern->std;
452: } /* }}} */
453:
454: static zend_object *spl_dlist_object_new(zend_class_entry *class_type) /* {{{ */
455: {
456:     return spl_dlist_object_new_ex(class_type, NULL, 0);
457: }
458: /* }}} */
459:
460: static zend_object *spl_dlist_object_clone(zval *zobject) /* {{{ */
461: {
462:     zend_object *old_object;
463:     zend_object *new_object;
464:
465:     old_object = Z_OBJ_P(zobject);
466:     new_object = spl_dlist_object_new_ex(old_object->ce, zobject, 1);
467:
468:     zend_objects_clone_members(new_object, old_object);
469:
470:     return new_object;
471: }
472: /* }}} */
473:
474: static int spl_dlist_object_count_elements(zval *zobject, zend_long *count) /* {{{ */
475: {
476:     spl_dlist_object *intern = Z_SPDOLLIST_P(zobject);
477:
478:     if (intern->fptr_count) {
479:         zend_call_method_with_0_params(object, intern->std.ce, intern->fptr_count, "count", &rv);
480:         if (!Z_ISNUMERIC(rv)) {
481:             *count = zval_get_long(&rv);
482:             *count = zval_get_long(&rv);
483:             return SUCCESS;
484:         }
485:         *count = 0;
486:         return FAILURE;
487:     }
488:
489:     *count = spl_ptr_llist_count(intern->llist);
490:     return SUCCESS;
491: }
492: /* }}} */
493:
494: static HashTable* spl_dlist_object_get_debug_info(zval *obj, int *is_a_temp) /* {{{ */
495: {
496:     spl_dlist_object *intern = Z_SPDOLLIST_P(obj);
497:     spl_ptr_llist_element *current = intern->llist->thead;
498:     spl_ptr_llist_element *next;
499:     zval tmp, dlist_array;
500:     zend_string *pnstr;
501:     int i = 0;
502:     HashTable *debug_info;
503:     *is_a_temp = 1;
504:
505:     if (!(&intern->std.properties)) {
506:         rebuild_object_properties(&intern->std);
507:     }
508:
509:     debug_info = zend_new_array(1);
510:     zend_hash_copy(debug_info, &intern->std.properties, (copy_ctor_func_t) zval_add_ref);
511:
512:     pnstr = spl_gen_private_prop_name(spl_oc_SplDoublyLinkedList, "Flags", sizeof("Flags")-1);
513:     ZVAL_LONG(&tmp, intern->flags);
514:     zend_hash_add(debug_info, pnstr, &tmp);
515:     zend_string_release(pnstr);
516:
517:     array_init(&dlist_array);
518:
519:     while (current) {
520:         next = current->next;
521:
522:         add_index_zval(&dlist_array, i, current->data);
523:         if (Z_REFCOUNTED(current->data)) {
524:             Z_ADDREF(current->data);
525:         }
526:         i++;
527:         current = next;
528:     }
529:
530:     pnstr = spl_gen_private_prop_name(spl_oc_SplDoublyLinkedList, "dlist", sizeof("dlist")-1);
531:     zend_hash_add(debug_info, pnstr, &dlist_array);
532:     zend_string_release(pnstr);
533:
534:     return debug_info;
535: }
536: /* }}} */
537:
538: static HashTable *spl_dlist_object_get_gc(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
539: {
540:     spl_dlist_object *intern = Z_SPDOLLIST_P(obj);
541:     spl_ptr_llist_element *current = intern->llist->thead;
542:     int i = 0;
543:
544:     if (intern->gc_data_count < intern->llist->count) {
545:         intern->gc_data_count = intern->llist->count;
546:         intern->gc_data = safe_erealloc(intern->gc_data, intern->gc_data_count, sizeof(zval), 0);
547:     }
548:
549:     while (current) {
550:         ZVAL_COPY_VALUE(intern->gc_data[i++], &current->data);
551:         current = current->next;
552:     }
553:
554:     *gc_data = intern->gc_data;
555:     *gc_data_count = i;
556:     return zend_std_get_properties(obj);
557: }
558: /* }}} */
559:
560: /* {{{ proto bool SplDoublyLinkedList::push(mixed value)
561: Push value on the SplDoublyLinkedList */
562: SPL_METHOD(SplDoublyLinkedList, push)
563: {
564:     zval *value;
565:     spl_dlist_object *intern;
566:
567:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &value) == FAILURE) {
568:         return;
569:     }
570:
571:     intern = Z_SPDOLLIST_P(getThis());
572:     spl_ptr_llist_push(intern->llist, value);
573:
574:     RETURN_TRUE;
575: }
576:
577: /* }}} */
578:
579: /* {{{ proto bool SplDoublyLinkedList::unshift(mixed value)
580: Unshift value on the SplDoublyLinkedList */
581: SPL_METHOD(SplDoublyLinkedList, unshift)
582: {
583:     zval *value;
584:     spl_dlist_object *intern;
585:
586:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "a", &value) == FAILURE) {
587:         return;
588:     }
589:
590:     intern = Z_SPDOLLIST_P(getThis());
591:     spl_ptr_llist_unshift(intern->llist, value);
592:
593:     RETURN_TRUE;
594: }
595:
596: /* }}} */
597:
598: /* {{{ proto mixed SplDoublyLinkedList::pop()
599: Pop an element out of the SplDoublyLinkedList */
600: SPL_METHOD(SplDoublyLinkedList, pop)
601: {
602:     spl_dlist_object *intern;
603:
604:     if (zend_parse_parameters_none() == FAILURE) {
605:         return;
606:     }
607:
608:     intern = Z_SPDOLLIST_P(getThis());
609:     spl_ptr_llist_pop(intern->llist, return_value);
610:
611:     if (Z_ISUNDEF_P(return_value)) {
612:         zend_throw_exception(spl_oc_RuntimeException, "Can't pop from an empty datastructure", 0);
613:         RETURN_NULL();
614:     }
615:
616:     /* {{{ proto mixed SplDoublyLinkedList::shift()
617: Shift an element out of the SplDoublyLinkedList */
618:     SPL_METHOD(SplDoublyLinkedList, shift)
619:     {
620:         spl_dlist_object *intern;
621:
622:         if (zend_parse_parameters_none() == FAILURE) {
623:             return;
624:         }
625:
626:         intern = Z_SPDOLLIST_P(getThis());
627:         spl_ptr_llist_shift(intern->llist, return_value);
628:
629:         if (Z_ISUNDEF_P(return_value)) {
630:             zend_throw_exception(spl_oc_RuntimeException, "Can't shift from an empty datastructure", 0);
631:             RETURN_NULL();
632:         }
633:
634:         /* }}} */
635:
636:     /* {{{ proto mixed SplDoublyLinkedList::top()
637: Peek at the top element of the SplDoublyLinkedList */
638:     SPL_METHOD(SplDoublyLinkedList, top)
639:     {
640:         zval *value;
641:         spl_dlist_object *intern;
642:
643:         if (zend_parse_parameters_none() == FAILURE) {
644:             return;
645:         }
646:
647:         intern = Z_SPDOLLIST_P(getThis());
648:         value = spl_ptr_llist_last(intern->llist);
649:
650:         if (value == NULL || Z_ISUNDEF_P(value)) {
651:             zend_throw_exception(spl_oc_RuntimeException, "Can't peek at an empty datastructure", 0);
652:             return;
653:         }
654:
655:         ZVAL_DEREF(value);
656:         ZVAL_COPY(return_value, value);
657:
658:         /* }}} */
659:
660:     /* {{{ proto mixed SplDoublyLinkedList::bottom()
661: Peek at the bottom element of the SplDoublyLinkedList */
662:     SPL_METHOD(SplDoublyLinkedList, bottom)
663:     {
664:         zval *value;
665:         spl_dlist_object *intern;
666:
667:         if (zend_parse_parameters_none() == FAILURE) {
668:             return;
669:         }
670:
671:         intern = Z_SPDOLLIST_P(getThis());
672:         value = spl_ptr_llist_first(intern->llist);
673:
674:         if (value == NULL || Z_ISUNDEF_P(value)) {
675:             zend_throw_exception(spl_oc_RuntimeException, "Can't peek at an empty datastructure", 0);
676:             return;
677:         }
678:
679:         ZVAL_DEREF(value);
680:         ZVAL_COPY(return_value, value);
681:
682:         /* }}} */
683:
684:     /* {{{ proto int SplDoublyLinkedList::count()
685: Return the number of elements in the datastructure. */
686:     SPL_METHOD(SplDoublyLinkedList, count)
687:     {
688:         zend_long count;
689:         spl_dlist_object *intern = Z_SPDOLLIST_P(getThis());
690:
691:         if (zend_parse_parameters_none() == FAILURE) {
692:             return;
693:         }
694:
695:         count = spl_ptr_llist_count(intern->llist);
696:         RETURN_LONG(count);
697:
698:         /* }}} */
699:
700:     /* {{{ proto int SplDoublyLinkedList::isEmpty()
701: Return true if the SplDoublyLinkedList is empty. */
702:     SPL_METHOD(SplDoublyLinkedList, isEmpty)
703:     {
704:         zend_long count;
705:
706:         if (zend_parse_parameters_none() == FAILURE) {
707:             return;
708:         }
709:
710:         spl_dlist_object_count_elements(getThis(), &count);
711:         RETURN_BOOL(count == 0);
712:
713:         /* }}} */
714:
715:     /* {{{ proto int SplDoublyLinkedList::setIteratorMode(int flags)
716: Set the mode of iteration */
717:     SPL_METHOD(SplDoublyLinkedList, setIteratorMode)
718:     {
719:         zend_long value;
720:         spl_dlist_object *intern;
721:
722:         if (zend_parse_parameters(ZEND_NUM_ARGS(), "i", &value) == FAILURE) {
723:             return;
724:         }
725:
726:         intern = Z_SPDOLLIST_P(getThis());
727:
728:         if (intern->flags & SPL_DLIST_IT_FIX) {
729:             if (intern->flags & SPL_DLIST_IT_IFPO) {
730:                 zend_throw_exception(spl_oc_RuntimeException, "Iterators' LIFO/FIFO modes for SplStack/SplQueue objects are frozen", 0);
731:                 return;
732:             }
733:
734:             intern->flags = (value & SPL_DLIST_IT_MASK) | (intern->flags & SPL_DLIST_IT_FIX);
735:
736:             RETURN_LONG(intern->flags);
737:
738:             /* }}} */
739:
740:         /* {{{ proto int SplDoublyLinkedList::getIteratorMode()
741: Return the mode of iteration */
742:         SPL_METHOD(SplDoublyLinkedList, getIteratorMode)
743:         {
744:             spl_dlist_object *intern;
745:
746:             if (zend_parse_parameters_none() == FAILURE) {
747:                 return;
748:             }
749:
750:             intern = Z_SPDOLLIST_P(getThis());
751:
752:

```

```

753: RETURN_LONG(intern->flags);
754: }
755: /* }}} */
756:
757: /* {{{ proto bool SplDoublyLinkedList::offsetExists(mixed index)
758: Returns whether the requested $index exists. */
759: SPL_METHOD(SplDoublyLinkedList, offsetExists)
760: {
761:     zval *zindex;
762:     spl_dlist_object *intern;
763:     zend_long index;
764:
765:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
766:         return;
767:     }
768:
769:     intern = Z_SPDLIST_P(getThis());
770:     index = spl_offset_convert_to_long(index);
771:
772:     RETURN_BOOL(index >= 0 && index < intern->vlist->count);
773: } /* }}} */
774:
775: /* {{{ proto mixed SplDoublyLinkedList::offsetGet(mixed index)
776: Returns the value at the specified $index. */
777: SPL_METHOD(SplDoublyLinkedList, offsetGet)
778: {
779:     zval *zindex;
780:     zend_long index;
781:     spl_dlist_object *intern;
782:     spl_ptr_list_element *element;
783:
784:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
785:         return;
786:     }
787:
788:     intern = Z_SPDLIST_P(getThis());
789:     index = spl_offset_convert_to_long(index);
790:
791:     if (index < 0 || index > intern->vlist->count) {
792:         zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid or out of range", 0);
793:         return;
794:     }
795:
796:     element = spl_ptr_list_offset(intern->vlist, index, intern->flags & SPL_DLIST_IT_LIFO);
797:
798:     if (element != NULL) {
799:         zval *value = element->data;
800:
801:         ZVAL_DEREF(value);
802:         ZVAL_COPY(return_value, value);
803:     } else {
804:         zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid", 0);
805:     }
806: } /* }}} */
807:
808: /* {{{ proto void SplDoublyLinkedList::offsetSet(mixed index, mixed value)
809: Sets the value at the specified $index to $value. */
810: SPL_METHOD(SplDoublyLinkedList, offsetSet)
811: {
812:     zval *zindex, *value;
813:     spl_dlist_object *intern;
814:
815:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "az", &zindex, &value) == FAILURE) {
816:         return;
817:     }
818:
819:     intern = Z_SPDLIST_P(getThis());
820:
821:     if (Z_TYPE_P(zindex) == IS_NULL) {
822:         /* $obj[] = ... */
823:         spl_ptr_list_push(intern->vlist, value);
824:     } else {
825:         /* $obj[$foo] = ... */
826:         zend_long index;
827:         spl_ptr_list_element *element;
828:
829:         index = spl_offset_convert_to_long(index);
830:
831:         if (index < 0 || index > intern->vlist->count) {
832:             zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid or out of range", 0);
833:             return;
834:         }
835:
836:         element = spl_ptr_list_offset(intern->vlist, index, intern->flags & SPL_DLIST_IT_LIFO);
837:
838:         if (element != NULL) {
839:             /* call dtor on the old element as in spl_ptr_list_pop */
840:             if (intern->vlist->dtor) {
841:                 intern->vlist->dtor(element);
842:             }
843:
844:             /* the element is replaced, deref the old one as in
845:              * SplDoublyLinkedList::pop() */
846:             zval_ptr_dtor(&element->data);
847:             ZVAL_COPY_VALUE(element->data, value);
848:
849:             /* new element, call ctor as in spl_ptr_list_push */
850:             if (intern->vlist->ctor) {
851:                 intern->vlist->ctor(element);
852:             }
853:         } else {
854:             zval_ptr_dtor(value);
855:             zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid", 0);
856:             return;
857:         }
858:     }
859: } /* }}} */
860:
861: /* {{{ proto void SplDoublyLinkedList::offsetUnset(mixed index)
862: Unsets the value at the specified $index. */
863: SPL_METHOD(SplDoublyLinkedList, offsetUnset)
864: {
865:     zval *zindex;
866:     zend_long index;
867:     spl_dlist_object *intern;
868:     spl_ptr_list_element *element;
869:     spl_ptr_list *vlist;
870:
871:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "z", &zindex) == FAILURE) {
872:         return;
873:     }
874:
875:     intern = Z_SPDLIST_P(getThis());
876:     index = spl_offset_convert_to_long(index);
877:     vlist = intern->vlist;
878:
879:     if (index < 0 || index > intern->vlist->count) {
880:         zend_throw_exception(spl_ce_OutOfRangeException, "Offset out of range", 0);
881:         return;
882:     }
883:
884:     element = spl_ptr_list_offset(intern->vlist, index, intern->flags & SPL_DLIST_IT_LIFO);
885:
886:     if (element != NULL) {
887:         /* connect the neighbors */
888:         if (element->prev) {
889:             element->prev->next = element->next;
890:         }
891:
892:         if (element->next) {
893:             element->next->prev = element->prev;
894:         }
895:
896:         /* take care of head/tail */
897:         if (element == vlist->head) {
898:             vlist->head = element->next;
899:         }
900:
901:         if (element == vlist->tail) {
902:             vlist->tail = element->prev;
903:         }
904:
905:         /* finally, delete the element */
906:         vlist->count--;
907:
908:         if (vlist->dtor) {
909:             vlist->dtor(element);
910:         }
911:
912:         if (intern->traverse_pointer == element) {
913:             SPL_LIST_DELREF(element);
914:             intern->traverse_pointer = NULL;
915:         }
916:         zval_ptr_dtor(&element->data);
917:         ZVAL_UNREF(element->data);
918:
919:         SPL_LIST_DELREF(element);
920:     } else {
921:         zend_throw_exception(spl_ce_OutOfRangeException, "Offset invalid", 0);
922:         return;
923:     }
924: } /* }}} */
925:
926: static void spl_dlist_it_dtor(zend_object_iterator *iter) /* {{{ */
927: {
928:     spl_dlist_it *iterator = (spl_dlist_it *)iter;
929:
930:     SPL_LIST_CHECK_DELREF(iterator->traverse_pointer);
931:
932:     zend_user_it_invalidate_current(iterator);
933:     zval_ptr_dtor(&iterator->intern->it_data);
934: }
935: /* }}} */
936:
937: static void spl_dlist_it_helper_rewind(spl_ptr_list_element **traverse_pointer_ptr, int *traverse_position_ptr, spl_ptr_list *vlist, int flags) /* {{{ */
938: {
939:     SPL_LIST_CHECK_DELREF(*traverse_pointer_ptr);
940:
941:     if (flags & SPL_DLIST_IT_LIFO) {
942:         *traverse_position_ptr = vlist->count-1;
943:         *traverse_pointer_ptr = vlist->tail;
944:     } else {
945:         *traverse_position_ptr = 0;
946:         *traverse_pointer_ptr = vlist->head;
947:     }
948:
949:     SPL_LIST_CHECK_ADDREF(*traverse_pointer_ptr);
950: }
951: /* }}} */
952:
953: static void spl_dlist_it_helper_move_forward(spl_ptr_list_element **traverse_pointer_ptr, int *traverse_position_ptr, spl_ptr_list *vlist, int flags) /* {{{ */
954: {
955:     if (*traverse_position_ptr) {
956:         spl_ptr_list_element *old = *traverse_pointer_ptr;
957:
958:         if (flags & SPL_DLIST_IT_LIFO) {
959:             *traverse_position_ptr = old->prev;
960:             (*traverse_position_ptr)--;
961:         }
962:
963:         if (flags & SPL_DLIST_IT_DELETE) {
964:             zval prev;
965:             spl_ptr_list_pop(vlist, &prev);
966:
967:             zval_ptr_dtor(&prev);
968:         } else {
969:             *traverse_position_ptr = old->next;
970:         }
971:
972:         if (flags & SPL_DLIST_IT_DELETE) {
973:             zval prev;
974:             spl_ptr_list_shift(vlist, &prev);
975:
976:             zval_ptr_dtor(&prev);
977:         } else {
978:             (*traverse_position_ptr)++;
979:         }
980:     }
981:
982:     SPL_LIST_CHECK_DELREF(old);
983:     SPL_LIST_CHECK_ADDREF(*traverse_pointer_ptr);
984: }
985: /* }}} */
986:
987: static void spl_dlist_it_rewind(zend_object_iterator *iter) /* {{{ */
988: {
989:     spl_dlist_it *iterator = (spl_dlist_it *)iter;
990:     spl_ptr_list_element *element = Z_SPDLIST_P(iterator->data);
991:     spl_ptr_list *vlist = element->vlist;
992:
993:     spl_dlist_it_helper_rewind(&iterator->traverse_pointer, &iterator->traverse_position, vlist, element->flags);
994: }
995: /* }}} */
996:
997: static int spl_dlist_it_valid(zend_object_iterator *iter) /* {{{ */
998: {
999:     spl_dlist_it *iterator = (spl_dlist_it *)iter;
1000:     spl_ptr_list_element *element = iterator->traverse_pointer;
1001:
1002:     return (element != NULL ? SUCCESS : FAILURE);
1003: }
1004: /* }}} */
1005:
1006: static zval *spl_dlist_it_get_current_data(zend_object_iterator *iter) /* {{{ */
1007: {
1008:     spl_dlist_it *iterator = (spl_dlist_it *)iter;
1009:     spl_ptr_list_element *element = iterator->traverse_pointer;
1010:
1011:     if (element == NULL || Z_ISUNDEF(element->data)) {
1012:         return NULL;
1013:     }
1014:
1015:     return element->data;
1016: }
1017: /* }}} */
1018:
1019: static void spl_dlist_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
1020: {
1021:     spl_dlist_it *iterator = (spl_dlist_it *)iter;
1022:
1023:     ZVAL_LONG(key, iterator->traverse_position);
1024: }
1025: /* }}} */
1026:
1027: static void spl_dlist_it_move_forward(zend_object_iterator *iter) /* {{{ */
1028: {
1029:     spl_dlist_it *iterator = (spl_dlist_it *)iter;
1030:     spl_ptr_list_element *object = Z_SPDLIST_P(iterator->data);
1031:
1032:     zend_user_it_invalidate_current(iter);
1033:
1034:     spl_dlist_it_helper_move_forward(&iterator->traverse_pointer, &iterator->traverse_position, object->vlist, object->flags);
1035: }
1036: /* }}} */
1037:
1038: /* {{{ proto int SplDoublyLinkedList::key()
1039: Return current array key */
1040: SPL_METHOD(SplDoublyLinkedList, key)
1041: {
1042:     spl_ptr_list_element *intern = Z_SPDLIST_P(getThis());
1043:
1044:     if (zend_parse_parameters_none() == FAILURE) {
1045:         return;
1046:     }
1047:
1048:     RETURN_LONG(intern->traverse_position);
1049: }
1050: /* }}} */
1051:
1052: /* {{{ proto void SplDoublyLinkedList::prev()
1053: Move to next entry */
1054: SPL_METHOD(SplDoublyLinkedList, prev)
1055: {
1056:     spl_ptr_list_element *intern = Z_SPDLIST_P(getThis());
1057:
1058:     if (zend_parse_parameters_none() == FAILURE) {
1059:         return;
1060:     }
1061:
1062:     spl_dlist_it_helper_move_forward(&intern->traverse_pointer, &intern->traverse_position, intern->vlist, intern->flags & SPL_DLIST_IT_LIFO);
1063: }
1064: /* }}} */
1065:
1066: /* {{{ proto void SplDoublyLinkedList::next()
1067: Move to next entry */
1068: SPL_METHOD(SplDoublyLinkedList, next)
1069: {
1070:     spl_ptr_list_element *intern = Z_SPDLIST_P(getThis());
1071:
1072:     if (zend_parse_parameters_none() == FAILURE) {
1073:         return;
1074:     }
1075:
1076:     spl_dlist_it_helper_move_forward(&intern->traverse_pointer, &intern->traverse_position, intern->vlist, intern->flags);
1077: }
1078: /* }}} */
1079:
1080: /* {{{ proto bool SplDoublyLinkedList::valid()
1081: Check whether the datastructure contains more entries */
1082: SPL_METHOD(SplDoublyLinkedList, valid)
1083: {
1084:     spl_ptr_list_element *intern = Z_SPDLIST_P(getThis());
1085:
1086:     if (zend_parse_parameters_none() == FAILURE) {
1087:         return;
1088:     }
1089:
1090:     RETURN_BOOL(intern->traverse_pointer != NULL);
1091: }
1092: /* }}} */
1093:
1094: /* {{{ proto void SplDoublyLinkedList::rewind()
1095: Rewind the datastructure back to the start */
1096: SPL_METHOD(SplDoublyLinkedList, rewind)
1097: {
1098:     spl_ptr_list_element *intern = Z_SPDLIST_P(getThis());
1099:
1100:     if (zend_parse_parameters_none() == FAILURE) {
1101:         return;
1102:     }
1103:
1104:     spl_dlist_it_helper_rewind(&intern->traverse_pointer, &intern->traverse_position, intern->vlist, intern->flags);
1105: }
1106: /* }}} */
1107:
1108: /* {{{ proto mixed SplDoublyLinkedList::current()
1109: Return current datastructure entry */
1110: SPL_METHOD(SplDoublyLinkedList, current)
1111: {
1112:     spl_ptr_list_element *intern = Z_SPDLIST_P(getThis());
1113:     spl_ptr_list_element *element = intern->traverse_pointer;
1114:
1115:     if (zend_parse_parameters_none() == FAILURE) {
1116:         return;
1117:     }
1118:
1119:     if (element == NULL || Z_ISUNDEF(element->data)) {
1120:         RETURN_NULL();
1121:     } else {
1122:         zval *value = element->data;
1123:
1124:         ZVAL_DEREF(value);
1125:         ZVAL_COPY(return_value, value);
1126:     }
1127: }

```

```

1127: }
1128: /* }}} */
1129:
1130: /* {{{ proto string SplDoublyLinkedList::serialize()
1131:  * Serializes storage */
1132: SPL_METHOD(SplDoublyLinkedList, serialize)
1133: {
1134:     spl_dlist_object *intern = Z_SPDLIST_P(getThis());
1135:     smart_str buf = (0);
1136:     spl_dlist_element *current = intern->llist->head, *next;
1137:     zval *zval;
1138:     php_serialize_data_t var_hash;
1139:
1140:     if (zend_parse_parameters_none() == FAILURE) {
1141:         return;
1142:     }
1143:
1144:     PHP_VAR_SERIALIZE_INIT(var_hash);
1145:
1146:     /* flags */
1147:     ZVAL_LONG(&flags, intern->flags);
1148:     php_var_serialize(&buf, &flags, &var_hash);
1149:     zval_get_ctor(&flags);
1150:
1151:     /* elements */
1152:     while (current) {
1153:         smart_str_append(&buf, ':');
1154:         next = current->next;
1155:
1156:         php_var_serialize(&buf, &current->data, &var_hash);
1157:
1158:         current = next;
1159:     }
1160:
1161:     smart_str_0(&buf);
1162:
1163:     /* done */
1164:     PHP_VAR_SERIALIZE_DESTROY(var_hash);
1165:
1166:     if (&buf.s) {
1167:         RETURN_NEW_STR(&buf.s);
1168:     } else {
1169:         RETURN_NULL();
1170:     }
1171: }
1172: /* }}} */
1173:
1174: /* {{{ proto void SplDoublyLinkedList::unserialize(string serialized)
1175:  * Unserializes storage */
1176: SPL_METHOD(SplDoublyLinkedList, unserialize)
1177: {
1178:     spl_dlist_object *intern = Z_SPDLIST_P(getThis());
1179:     zval *flags, *elem;
1180:     char *buf;
1181:     elem->buf_len;
1182:     const unsigned char *p, *s;
1183:     php_unserialize_data_t var_hash;
1184:
1185:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "s", &buf, &buf_len) == FAILURE) {
1186:         return;
1187:     }
1188:
1189:     if (&buf_len == 0) {
1190:         return;
1191:     }
1192:
1193:     s = p = (const unsigned char *)buf;
1194:     PHP_VAR_UNSERIALIZE_INIT(var_hash);
1195:
1196:     /* flags */
1197:     flags = var_tmp_var(var_hash);
1198:     if (!php_var_unserialize(&flags, &p, s + &buf_len, &var_hash) || Z_TYPE_P(flags) != IS_LONG) {
1199:         goto error;
1200:     }
1201:
1202:     intern->flags = (int)Z_LVAL_P(flags);
1203:
1204:     /* elements */
1205:     while (*p != ':') {
1206:         *p;
1207:         elem = var_tmp_var(var_hash);
1208:         if (!php_var_unserialize(&elem, &p, s + &buf_len, &var_hash)) {
1209:             goto error;
1210:         }
1211:         var_push_ctor(&var_hash, elem);
1212:
1213:         spl_dlist_push(intern->llist, elem);
1214:     }
1215:
1216:     if (*p != '\0') {
1217:         goto error;
1218:     }
1219:
1220:     PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
1221:
1222:     return;
1223: error:
1224:     PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
1225:     zend_throw_exception_ex(spl_ce UnexpectedValueException, 0, "Error at offset %d of %d bytes", ((char *)p - buf), &buf_len);
1226:     return;
1227: }
1228: /* }}} */
1229:
1230: /* {{{ proto void SplDoublyLinkedList::add(mixed index, mixed newval)
1231:  * Inserts a new entry before the specified index consisting of $newval. */
1232: SPL_METHOD(SplDoublyLinkedList, add)
1233: {
1234:     zval *index, *value;
1235:     spl_dlist_object *intern;
1236:     spl_dlist_element *element;
1237:     zend_long index;
1238:
1239:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "zs", &index, &value) == FAILURE) {
1240:         return;
1241:     }
1242:
1243:     intern = Z_SPDLIST_P(getThis());
1244:     index = spl_offset_convert_to_long(index);
1245:
1246:     if (index < 0 || index > intern->llist->count) {
1247:         zend_throw_exception(spl_ce OutOfRangeException, "Offset invalid or out of range", 0);
1248:         return;
1249:     }
1250:
1251:     Z_TRY_ADDREF_P(value);
1252:
1253:     if (index == intern->llist->count) {
1254:         /* If index is the last element then we do a push because we're not inserting before any entry */
1255:         spl_dlist_push(intern->llist, value);
1256:     } else {
1257:         /* Create the new element we want to insert */
1258:         spl_dlist_element *elem = emalloc(sizeof(spl_dlist_element));
1259:
1260:         /* Get the element we want to insert before */
1261:         element = spl_dlist_offset(intern->llist, index, intern->flags & SPL_DLIST_IT_LIFO);
1262:
1263:         ZVAL_COPY_VALUE(&elem->data, value);
1264:         elem->vc = 1;
1265:         /* connect to the neighbours */
1266:         elem->next = element;
1267:         elem->prev = element->prev;
1268:
1269:         /* connect the neighbours to this new element */
1270:         if (elem->prev == NULL) {
1271:             intern->llist->head = elem;
1272:         } else {
1273:             element->prev->next = elem;
1274:         }
1275:
1276:         intern->llist->count++;
1277:
1278:         if (intern->llist->ctor) {
1279:             intern->llist->ctor(elem);
1280:         }
1281:     }
1282: }
1283: /* }}} */
1284:
1285: /* {{{ iterator handler table */
1286: static const zend_object_iterator_funcs spl_dlist_it_funcs = {
1287:     spl_dlist_it_ctor,
1288:     spl_dlist_it_valid,
1289:     spl_dlist_it_get_current_data,
1290:     spl_dlist_it_get_current_key,
1291:     spl_dlist_it_move_forward,
1292:     spl_dlist_it_rewind,
1293:     NULL,
1294: };
1295: /* }}} */
1296:
1297: zend_object_iterator *spl_dlist_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1298: {
1299:     spl_dlist_it *iterator;
1300:     spl_dlist_object *dlist_object = Z_SPDLIST_P(object);
1301:
1302:     if (by_ref) {
1303:         zend_throw_exception(spl_ce RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1304:         return NULL;
1305:     }
1306:
1307:     iterator = emalloc(sizeof(spl_dlist_it));
1308:
1309:     zend_iterator_init(&(zend_object_iterator *)iterator);
1310:
1311:     ZVAL_COPY_VALUE(&iterator->intern.it.data, object);
1312:     iterator->intern.it.funcs = &spl_dlist_it_funcs;
1313:     iterator->intern.ce = ce;
1314:     iterator->traverse_position = dlist_object->traverse_position;
1315:     iterator->traverse_pointer = dlist_object->traverse_pointer;
1316:     iterator->flags = dlist_object->flags & SPL_DLIST_IT_MASK;
1317: }
1318:
1319: ZVAL_UNDEF(iterator->intern.value);
1320:
1321: SPL_LIST_CHECK_ADDRESS(iterator->traverse_pointer);
1322:
1323: return iterator->intern.it;
1324: }
1325:
1326: /* Functions/Class/Method definitions */
1327:
1328: ZEND_BEGIN_ARG_INFO(arginfo_dlist_set_iterator_mode, 0)
1329:     ZEND_ARG_INFO(0, flags)
1330: ZEND_END_ARG_INFO()
1331:
1332: ZEND_BEGIN_ARG_INFO(arginfo_dlist_push, 0)
1333:     ZEND_ARG_INFO(0, value)
1334: ZEND_END_ARG_INFO()
1335:
1336: ZEND_BEGIN_ARG_INFO(arginfo_dlist_offset_get, 0, 0, 1)
1337:     ZEND_ARG_INFO(0, index)
1338: ZEND_END_ARG_INFO()
1339:
1340: ZEND_BEGIN_ARG_INFO(arginfo_dlist_offset_set, 0, 0, 2)
1341:     ZEND_ARG_INFO(0, index)
1342:     ZEND_ARG_INFO(0, newval)
1343: ZEND_END_ARG_INFO()
1344:
1345: ZEND_BEGIN_ARG_INFO(arginfo_dlist_void, 0)
1346:     ZEND_ARG_INFO(0, serialized)
1347: ZEND_END_ARG_INFO()
1348:
1349: static const zend_function_entry spl_funcs_SplQueue[] = {
1350:     SPL_METHOD(SplQueue, enqueue, SplDoublyLinkedList, push, arginfo_dlist_push, ZEND_ACC_PUBLIC)
1351:     SPL_METHOD(SplQueue, dequeue, SplDoublyLinkedList, shift, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1352:     SPL_FE_END
1353: };
1354:
1355: static const zend_function_entry spl_funcs_SplDoublyLinkedList[] = {
1356:     SPL_METHOD(SplDoublyLinkedList, pop, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1357:     SPL_METHOD(SplDoublyLinkedList, shift, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1358:     SPL_METHOD(SplDoublyLinkedList, push, arginfo_dlist_push, ZEND_ACC_PUBLIC)
1359:     SPL_METHOD(SplDoublyLinkedList, unshift, arginfo_dlist_push, ZEND_ACC_PUBLIC)
1360:     SPL_METHOD(SplDoublyLinkedList, top, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1361:     SPL_METHOD(SplDoublyLinkedList, bottom, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1362:     SPL_METHOD(SplDoublyLinkedList, isEmpty, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1363:     SPL_METHOD(SplDoublyLinkedList, setIteratorMode, arginfo_dlist_set_iterator_mode, ZEND_ACC_PUBLIC)
1364:     SPL_METHOD(SplDoublyLinkedList, getIteratorMode, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1365:     /* Countable */
1366:     SPL_METHOD(SplDoublyLinkedList, count, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1367:     /* ArrayAccess */
1368:     SPL_METHOD(SplDoublyLinkedList, offsetExists, arginfo_dlist_offset_get, ZEND_ACC_PUBLIC)
1369:     SPL_METHOD(SplDoublyLinkedList, offsetGet, arginfo_dlist_offset_get, ZEND_ACC_PUBLIC)
1370:     SPL_METHOD(SplDoublyLinkedList, offsetSet, arginfo_dlist_offset_get, ZEND_ACC_PUBLIC)
1371:     SPL_METHOD(SplDoublyLinkedList, offsetUnset, arginfo_dlist_offset_get, ZEND_ACC_PUBLIC)
1372:     SPL_METHOD(SplDoublyLinkedList, add, arginfo_dlist_offset_set, ZEND_ACC_PUBLIC)
1373:
1374:     /* Iterator */
1375:     SPL_METHOD(SplDoublyLinkedList, rewind, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1376:     SPL_METHOD(SplDoublyLinkedList, current, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1377:     SPL_METHOD(SplDoublyLinkedList, key, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1378:     SPL_METHOD(SplDoublyLinkedList, next, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1379:     SPL_METHOD(SplDoublyLinkedList, prev, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1380:     SPL_METHOD(SplDoublyLinkedList, valid, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1381:     /* Serializable */
1382:     SPL_METHOD(SplDoublyLinkedList, unserialize, arginfo_dlist_serialized, ZEND_ACC_PUBLIC)
1383:     SPL_METHOD(SplDoublyLinkedList, serialize, arginfo_dlist_void, ZEND_ACC_PUBLIC)
1384:     SPL_FE_END
1385: };
1386: /* }}} */
1387:
1388: PHP_MINIT_FUNCTION(spl_dlist) /* {{{ */
1389: {
1390:     REGISTER_SPL_STD_CLASS_EX(SplDoublyLinkedList, spl_dlist_object_new, spl_funcs_SplDoublyLinkedList);
1391:     memory(spl_handler_SplDoublyLinkedList, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1392:
1393:     spl_handler_SplDoublyLinkedList.offset = XOffsetOf(spl_dlist_object, std);
1394:     spl_handler_SplDoublyLinkedList.clone_obj = spl_dlist_object_clone;
1395:     spl_handler_SplDoublyLinkedList.count_elements = spl_dlist_object_count_elements;
1396:     spl_handler_SplDoublyLinkedList.get_debug_info = spl_dlist_object_get_debug_info;
1397:     spl_handler_SplDoublyLinkedList.get_gc = spl_dlist_object_get_gc;
1398:     spl_handler_SplDoublyLinkedList.dtor_obj = zend_objects_destroy_object;
1399:     spl_handler_SplDoublyLinkedList.free_obj = spl_dlist_object_free_storage;
1400:
1401:     REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_LIFO", SPL_DLIST_IT_LIFO);
1402:     REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_FIFO", 0);
1403:     REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_DELETE", SPL_DLIST_IT_DELETE);
1404:     REGISTER_SPL_CLASS_CONST_LONG(SplDoublyLinkedList, "IT_MODE_KEEP", 0);
1405:
1406:     REGISTER_SPL_IMPLEMENT(SplDoublyLinkedList, Iterator);
1407:     REGISTER_SPL_IMPLEMENT(SplDoublyLinkedList, Countable);
1408:     REGISTER_SPL_IMPLEMENT(SplDoublyLinkedList, ArrayAccess);
1409:     REGISTER_SPL_IMPLEMENT(SplDoublyLinkedList, Serializable);
1410:
1411:     spl_ce_SplDoublyLinkedList->get_iterator = spl_dlist_get_iterator;
1412:
1413:     REGISTER_SPL_STD_CLASS_EX(SplQueue, SplDoublyLinkedList, spl_dlist_object_new, spl_funcs_SplQueue);
1414:     REGISTER_SPL_STD_CLASS_EX(SplStack, SplDoublyLinkedList, spl_dlist_object_new, NULL);
1415:
1416:     spl_ce_SplQueue->get_iterator = spl_dlist_get_iterator;
1417:     spl_ce_SplStack->get_iterator = spl_dlist_get_iterator;
1418:
1419:     return SUCCESS;
1420: }
1421: /* }}} */
1422:
1423: /*
1424:  * Local variables:
1425:  * tab-width: 4
1426:  * indent-offset: 4
1427:  * End:
1428:  * vim600: fdm=marker
1429:  * vim: noet sw=4 ts=4
1430:  */

```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: #ifdef HAVE_CONFIG_H
20: # include "config.h"
21: #endif
22:
23: #include "php.h"
24: #include "php_ini.h"
25: #include "ext/standard/info.h"
26: #include "zend_interfaces.h"
27:
28: #include "spl_spl.h"
29: #include "spl_functions.h"
30: #include "spl_engine.h"
31:
32: #include "spl_array.h"
33:
34: /* {{{ spl_initialize */
35: PHPAPI void spl_initialize(zend_class_entry *ce, zval *object)
36: {
37:     object_init_ex(object, ce);
38: }
39: /* }}} */
40:
41: PHPAPI zend_long spl_offset_convert_to_long(zval *offset) /* {{{ */
42: {
43:     zend_ulong idx;
44:
45:     try_again:
46:     switch (Z_TYPE_P(offset)) {
47:         case IS_STRING:
48:             if (ZEND_HANDLE_NUMERIC(Z_STR_P(offset), idx)) {
49:                 return idx;
50:             }
51:             break;
52:         case IS_DOUBLE:
53:             return (zend_long)Z_DVAL_P(offset);
54:         case IS_LONG:
55:             return Z_LVAL_P(offset);
56:         case IS_FALSE:
57:             return 0;
58:         case IS_TRUE:
59:             return 1;
60:         case IS_REFERENCE:
61:             offset = Z_REFVAL_P(offset);
62:             goto try_again;
63:         case IS_RESOURCE:
64:             return Z_RES_HANDLE_P(offset);
65:     }
66:     return -1;
67: }
68: /* }}} */
69:
70: /*
71:  * Local Variables:
72:  * tab-width: 4
73:  * c-basic-offset: 4
74:  * End:
75:  * vim600: fdm=marker
76:  * vim: noet sw=4 ts=4
77:  */
```

```

1: /*
2:  *
3:  * PHP Version 7
4:  *
5:  * Copyright (c) 1997-2018 The PHP Group
6:  *
7:  * This source file is subject to version 3.01 of the PHP license,
8:  * that is bundled with this package in the file LICENSE, and is
9:  * available through the world-wide-web at the following url:
10:  * http://www.php.net/license/3.01.txt
11:  * If you did not receive a copy of the PHP license and are unable to
12:  * obtain it through the world-wide-web, please send a note to
13:  * license@php.net so we can mail you a copy immediately.
14:  *
15:  * Author: Marcus Boerger <chelly@php.net>
16:  */
17:
18: /* $Id$ */
19:
20: #ifndef HAVE_CONFIG_H
21: #include "config.h"
22: #endif
23:
24: #include "php.h"
25: #include "php_ini.h"
26: #include "ext/standard/info.h"
27: #include "ext/standard/file.h"
28: #include "ext/standard/php_string.h"
29: #include "zend_compile.h"
30: #include "zend_exceptions.h"
31: #include "zend_interfaces.h"
32:
33: #include "php_spl.h"
34: #include "spl_functions.h"
35: #include "spl_engine.h"
36: #include "spl_iterators.h"
37: #include "spl_directory.h"
38: #include "spl_exceptions.h"
39:
40: #include "php.h"
41: #include "fopen_wrappers.h"
42:
43: #include "ext/standard/basic_functions.h"
44: #include "ext/standard/php_filestat.h"
45:
46: #define SPL_HAS_FLAG(flags, test_flag) ((flags & test_flag) ? 1 : 0)
47:
48: /* declare the class handlers */
49: static zend_object_handlers spl_filesystem_object_handlers;
50: /* Includes handler to validate object state when retrieving methods */
51: static zend_object_handlers spl_filesystem_object_check_handlers;
52:
53: /* declare the class entry */
54: PHPAPI zend_class_entry *spl_ce_SplFileInfo;
55: PHPAPI zend_class_entry *spl_ce_SplDirectoryIterator;
56: PHPAPI zend_class_entry *spl_ce_SplFilesystemIterator;
57: PHPAPI zend_class_entry *spl_ce_SplRecursiveDirectoryIterator;
58: PHPAPI zend_class_entry *spl_ce_SplGlobIterator;
59: PHPAPI zend_class_entry *spl_ce_SplFileObject;
60: PHPAPI zend_class_entry *spl_ce_SplTempFileObject;
61:
62: static void spl_filesystem_file_free_line(spl_filesystem_object *intern) /* {{{ */
63: {
64:     if (intern->u.file.current_line) {
65:         efree(intern->u.file.current_line);
66:         intern->u.file.current_line = NULL;
67:     }
68:     if (IS_INDEXED(intern->u.file.current_rval)) {
69:         eval_ptr_dtor(intern->u.file.current_rval);
70:         ZVAL_UNDEF(intern->u.file.current_rval);
71:     }
72: }
73: /* }}} */
74:
75: static void spl_filesystem_object_destroy_object(spl_filesystem_object *intern) /* {{{ */
76: {
77:     spl_filesystem_object *intern = spl_filesystem_from_obj(object);
78:     zend_object_dtor(object);
79: }
80:
81: switch (intern->type) {
82:     case SPL_FS_DIR:
83:         if (intern->u.dir.dirp) {
84:             php_stream_close(intern->u.dir.dirp);
85:             intern->u.dir.dirp = NULL;
86:         }
87:         break;
88:     case SPL_FS_FILE:
89:         if (intern->u.file.stream) {
90:             /*
91:              * If (intern->u.file.zcontext) {
92:              *     zend_list_delref(Z_RES_VAL_P(intern->zcontext));
93:              * }
94:              */
95:             if ((intern->u.file.stream->is_persistent) {
96:                 php_stream_close(intern->u.file.stream);
97:             } else {
98:                 php_stream_close(intern->u.file.stream);
99:             }
100:         }
101:         break;
102:     default:
103:         break;
104: }
105: /* }}} */
106:
107: static void spl_filesystem_object_free_storage(zend_object *object) /* {{{ */
108: {
109:     spl_filesystem_object *intern = spl_filesystem_from_obj(object);
110:     if (intern->both_handler && intern->both_handler->dtor) {
111:         intern->both_handler->dtor(intern);
112:     }
113:     zend_object_std_dtor(intern->std);
114: }
115:
116: if (intern->path) {
117:     efree(intern->path);
118: }
119: if (intern->file_name) {
120:     efree(intern->file_name);
121: }
122: switch (intern->type) {
123:     case SPL_FS_INFO:
124:         break;
125:     case SPL_FS_DIR:
126:         if (intern->u.dir.sub_path) {
127:             efree(intern->u.dir.sub_path);
128:         }
129:         break;
130:     case SPL_FS_FILE:
131:         if (intern->u.file.stream) {
132:             if (intern->u.file.open_mode) {
133:                 efree(intern->u.file.open_mode);
134:             }
135:             if (intern->u.orig_path) {
136:                 efree(intern->u.orig_path);
137:             }
138:         }
139:         break;
140:     default:
141:         break;
142: }
143: /* }}} */
144:
145: /* {{{ spl_dir_object_new */
146: /* creates the object by
147:  * - allocating memory
148:  * - initializing the object members
149:  * - storing the object
150:  * - setting its handlers
151:  */
152: /* called from
153:  * - clone
154:  * - new
155:  */
156: static zend_object *spl_filesystem_object_new_ex(zend_class_entry *class_type)
157: {
158:     spl_filesystem_object *intern;
159:     intern = zend_object_alloc(sizeof(spl_filesystem_object), class_type);
160:     /* intern->type = SPL_FS_INFO; done by set 0 */
161:     intern->file_class = spl_ce_SplFileInfo;
162:     intern->info_class = spl_ce_SplFileInfo;
163:     zend_object_std_init(intern->std, class_type);
164:     object_properties_init(intern->std, class_type);
165:     intern->std.handlers = spl_filesystem_object_check_handlers;
166:     return intern->std;
167: }
168: /* }}} */
169:
170: /* {{{ spl_filesystem_object_new */
171: /* See spl_filesystem_object_new_ex */
172: static zend_object *spl_filesystem_object_new(zend_class_entry *class_type)
173: {
174:     return spl_filesystem_object_new_ex(class_type);
175: }
176:
177: /* }}} */
178:
179: /* {{{ spl_filesystem_object_new_check */
180: static zend_object *spl_filesystem_object_new_check(zend_class_entry *class_type)
181: {
182:     spl_filesystem_object *ret = spl_filesystem_from_obj(spl_filesystem_object_new_ex(class_type));
183:     ret->std.handlers = spl_filesystem_object_check_handlers;
184:     return ret->std;
185: }
186: /* }}} */
187:
188:
189: PHPAPI char *spl_filesystem_object_get_path(spl_filesystem_object *intern, size_t *len) /* {{{ */
190: {
191:     if (intern->type == SPL_FS_DIR) {
192:         if (php_stream_is(intern->u.dir.dirp, &php_glob_stream_ops) {
193:             return php_glob_stream_get_path(intern->u.dir.dirp, 0, len);
194:         }
195:     }
196:     if (intern->type == SPL_FS_FILE) {
197:         if (intern->u.file.stream) {
198:             return php_stream_get_path(intern->u.file.stream, 0, len);
199:         }
200:     }
201:     return NULL;
202: }
203: /* }}} */
204:
205: static inline void spl_filesystem_object_get_file_name(spl_filesystem_object *intern) /* {{{ */
206: {
207:     char slash = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_UNIPATHS) ? '/' : DEFAULT_SLASH;
208:     switch (intern->type) {
209:         case SPL_FS_INFO:
210:             break;
211:         case SPL_FS_FILE:
212:             if (intern->file_name) {
213:                 php_error_docref(NULL, E_ERROR, "Object not initialized");
214:             }
215:             break;
216:         case SPL_FS_DIR:
217:             if (intern->file_name) {
218:                 efree(intern->file_name);
219:             }
220:             intern->file_name_len = spprintf(&intern->file_name, 0, "%s",
221:                 spl_filesystem_object_get_path(intern, NULL),
222:                 slash, intern->u.dir.entry_d_name);
223:             break;
224:     }
225: }
226: /* }}} */
227:
228: static int spl_filesystem_dir_read(spl_filesystem_object *intern) /* {{{ */
229: {
230:     if (intern->u.dir.dirp || !php_stream_readdir(intern->u.dir.dirp, &intern->u.dir.entry)) {
231:         return 0;
232:     }
233:     return 1;
234: }
235: /* }}} */
236:
237: #define IS_SLASH_AT(pos) (IS_SLASH(pos[pos]))
238:
239: static inline int spl_filesystem_is_dot(const char *d_name) /* {{{ */
240: {
241:     return !strcmp(d_name, ".") || !strcmp(d_name, "..");
242: }
243: /* }}} */
244:
245: /* {{{ spl_filesystem_dir_open */
246: /* open a directory resource */
247: static void spl_filesystem_dir_open(spl_filesystem_object *intern, char *path)
248: {
249:     int skip_dots = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_SKIPDOTS);
250:     intern->type = SPL_FS_DIR;
251:     intern->u.path_len = strlen(path);
252:     intern->u.dir.dirp = php_stream_opendir(path, REPORT_ERRORS, PG(default_context));
253:     if (intern->u.path_len > 1 && IS_SLASH_AT(path, intern->u.path_len-1)) {
254:         intern->u.path = estrndup(path, intern->u.path_len);
255:     } else {
256:         intern->u.path = estrndup(path, intern->u.path_len);
257:     }
258:     intern->u.dir.index = 0;
259: }
260:
261: if (IS_EXCEPTION) || intern->u.dir.dirp == NULL {
262:     intern->u.dir.entry_d_name[0] = '\0';
263:     if (IS_EXCEPTION) {
264:         /* open failed w/out notice (turned to exception due to E_THROW) */
265:         zend_throw_exception(spl_ce_RuntimeException, 0, "Failed to open directory \"%s\"", path);
266:     }
267:     return FAILURE;
268: }
269:
270: do {
271:     spl_filesystem_dir_read(intern);
272:     while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name)) {
273:         spl_filesystem_dir_read(intern);
274:     }
275: } while (1);
276: /* }}} */
277:
278: static int spl_filesystem_file_open(spl_filesystem_object *intern, int use_include_path, int silent) /* {{{ */
279: {
280:     zend_tmp;
281:     intern->type = SPL_FS_FILE;
282:     php_stat(intern->file_name, intern->file_name_len, FS_IS_DIR, &tmp);
283:     if (IS_TRUE(tmp) == IS_TRUE) {
284:         intern->u.file.open_mode = NULL;
285:         intern->file_name = NULL;
286:         zend_throw_exception(spl_ce_LogicException, 0, "Cannot use SplFileObject with directories");
287:         return FAILURE;
288:     }
289:     intern->u.file.zcontext = php_stream_context_from_rval(intern->u.file.zcontext, 0);
290:     intern->u.file.stream = php_stream_open_wrapper_ex(intern->file_name, intern->u.file.open_mode, (use_include_path ? USE_PATH : 0), REPORT_ERRORS, MU
291:     );
292:     if (intern->file_name_len != 1 && IS_SLASH_AT(intern->file_name, intern->file_name_len-1)) {
293:         if (IS_EXCEPTION) {
294:             zend_throw_exception(spl_ce_RuntimeException, 0, "Cannot open file \"%s\", intern->file_name_len ? intern->file_name : "");
295:         }
296:         intern->file_name = NULL; /* until here it is not a copy */
297:         intern->u.file.open_mode = NULL;
298:         return FAILURE;
299:     }
300:     /*
301:      * If (intern->u.file.zcontext) {
302:      *     zend_list_addref(Z_RES_VAL(intern->u.file.zcontext));
303:      * }
304:      */
305:     if (intern->file_name_len > 1 && IS_SLASH_AT(intern->file_name, intern->file_name_len-1)) {
306:         intern->u.orig_path = estrndup(intern->u.file.stream->orig_path, strlen(intern->u.file.stream->orig_path));
307:         intern->file_name = estrndup(intern->file_name, intern->file_name_len);
308:         intern->u.file.open_mode = estrndup(intern->u.file.open_mode, intern->u.file.open_mode_len);
309:     }
310:     /* avoid reference counting in debug mode, thus do it manually */
311:     ZVAL_RES(intern->u.file.resource, intern->u.file.stream->res);
312:     /*!!! TODO: maybe bug?
313:     zend_list_addref(intern->u.file.resource, 1);
314:     */
315:     intern->u.file.delimiter = '/';
316:     intern->u.file.enclosure = '';
317:     intern->u.file.escape = '\\';
318:     intern->u.file.func_getCurr = zend_hash_str_find_ptr(intern->std.ce->function_table, "getcurrenttime", sizeof("getcurrenttime") - 1);
319:     return SUCCESS;
320: }
321: /* }}} */
322:
323: /* {{{ spl_filesystem_object_clone */
324: /* Local zend_object creation (on stack)
325:  * Load the 'other' object
326:  * Create a new empty object (See spl_filesystem_object_new_ex)
327:  * Open the directory
328:  * Clone other members (properties)
329:  */
330: static zend_object *spl_filesystem_object_clone(zval *obj)
331: {
332:     zend_object *old_obj;
333:     zend_object *new_obj;
334:     spl_filesystem_object *intern;
335:     spl_filesystem_object *source;
336:     int index, skip_dots;
337:     old_obj = Z_OBJ_P(obj);
338:     source = spl_filesystem_from_obj(old_obj);
339:     new_obj = spl_filesystem_object_new_ex(old_obj->ce);
340:     intern = spl_filesystem_from_obj(new_obj);
341:     intern->flags = source->flags;
342:     switch (source->type) {
343:         case SPL_FS_INFO:
344:             intern->u.path_len = source->u.path_len;
345:             intern->u.path = estrndup(source->u.path, source->u.path_len);
346:             intern->file_name_len = source->file_name_len;
347:             intern->file_name = estrndup(source->file_name, intern->file_name_len);
348:             break;
349:         case SPL_FS_DIR:
350:             spl_filesystem_dir_open(intern, source->u.path);
351:             /* read until we hit the position in which we were before */
352:             skip_dots = SPL_HAS_FLAG(source->flags, SPL_FILE_DIR_SKIPDOTS);
353:             for (index = 0; index < source->u.dir.index; ++index) {
354:                 do {
355:                     spl_filesystem_dir_read(intern);
356:                     while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_d_name)) {
357:                         spl_filesystem_dir_read(intern);
358:                     }
359:                     intern->u.dir.index = index;
360:                     break;
361:                 } while (1);
362:             }
363:             break;
364:         case SPL_FS_FILE:
365:             break;
366:     }
367: }
368:
369:
370:
371:
372:
373:
374:
375:

```

```

376:     zend_throw_error(NULL, "An object of class is cannot be cloned", ZSTR_VAL(oid_object->ce->name));
377:     return new_object;
378: }
379:
380: intern->file_class = source->file_class;
381: intern->info_class = source->info_class;
382: intern->path = source->path;
383: intern->path_handler = source->path_handler;
384:
385: zend_object_clone_members(new_object, oid_object);
386:
387: IF (intern->path_handler && intern->path_handler->clone) {
388:     intern->path_handler->clone(source, intern);
389: }
390:
391: return new_object;
392: }
393: /* }}} */
394:
395: void spl_filesystem_info_set_filename(spl_filesystem_object *intern, char *path, size_t len, size_t use_copy) /* {{{ */
396: {
397:     char *p1, *p2;
398:
399:     IF (intern->file_name) {
400:         efree(intern->file_name);
401:     }
402:
403:     intern->file_name = use_copy ? estrndup(path, len) : path;
404:     intern->file_name_len = len;
405:
406:     while (intern->file_name_len > 1 && IS_SLASH_AT(intern->file_name, intern->file_name_len-1)) {
407:         intern->file_name[intern->file_name_len-1] = 0;
408:         intern->file_name_len--;
409:     }
410:
411:     p1 = strchr(intern->file_name, '/');
412:     IF (defined(PHP_WIN32))
413:         p2 = strchr(intern->file_name, '\\');
414:     else
415:         p2 = 0;
416:     IF (p1 || p2) {
417:         intern->path_len = ((p1 > p2 ? p1 : p2) - intern->file_name);
418:     } else {
419:         intern->path_len = 0;
420:     }
421:
422:     IF (intern->path) {
423:         efree(intern->path);
424:     }
425:
426:     intern->path = estrndup(path, intern->path_len);
427: } /* }}} */
428:
429: static spl_filesystem_object *spl_filesystem_object_create_info(spl_filesystem_object *source, char *file_name, size_t file_path_len, int use_copy, zend_class_entry *ce, zval *return_value) /* {{{ */
430: {
431:     spl_filesystem_object *intern;
432:     zval arg1;
433:     zend_error_handling error_handling;
434:
435:     IF (file_path || file_path_len) {
436:         IF (defined(PHP_WIN32))
437:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Cannot create SplFileInfo for empty path");
438:         IF (file_path && !use_copy) {
439:             efree(file_path);
440:         }
441:         else
442:             IF (file_path && !use_copy) {
443:                 efree(file_path);
444:             }
445:             file_path_len = 1;
446:             file_path = "/";
447:         IF (defined(PHP_WIN32))
448:             return NULL;
449:     }
450:
451:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, error_handling);
452:
453:     ce = ce ? ce : source->info_class;
454:
455:     zend_update_class_constants(ce);
456:
457:     intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
458:     ZVAL_OBJ(return_value, &intern->std);
459:
460:     IF (ce->constructor->common.scope != spl_ce_SplFileInfo) {
461:         ZVAL_STRING(&arg1, file_path, file_path_len);
462:         zend_call_method_with_1_param(return_value, ce, ce->constructor, "_construct", NULL, &arg1);
463:         zval_ptr_dtor(&arg1);
464:     } else {
465:         spl_filesystem_info_set_filename(intern, file_path, file_path_len, use_copy);
466:     }
467:
468:     zend_restore_error_handling(error_handling);
469:     return intern;
470: } /* }}} */
471:
472: static spl_filesystem_object *spl_filesystem_object_create_type(int ht, spl_filesystem_object *source, int type, zend_class_entry *ce, zval *return_value) /* {{{ */
473: {
474:     spl_filesystem_object *intern;
475:     zend_bool use_include_path = 0;
476:     zval arg1, arg2;
477:     zend_error_handling error_handling;
478:
479:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, error_handling);
480:
481:     switch (source->type) {
482:         case SPL_FS_THROW:
483:             case SPL_FS_FILE:
484:                 break;
485:             case SPL_FS_DIR:
486:                 IF (!source->v.dir.entry.d_name[0]) {
487:                     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Could not open file");
488:                     zend_restore_error_handling(error_handling);
489:                     return NULL;
490:                 }
491:             }
492:
493:     switch (type) {
494:         case SPL_FS_THROW:
495:             ce = ce ? ce : source->info_class;
496:
497:             IF (UNEXPECTED(zend_update_class_constants(ce) != SUCCESS)) {
498:                 break;
499:             }
500:
501:             intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
502:             ZVAL_OBJ(return_value, &intern->std);
503:
504:             spl_filesystem_object_get_file_name(source);
505:             IF (ce->constructor->common.scope != spl_ce_SplFileInfo) {
506:                 ZVAL_STRING(&arg1, source->file_name, source->file_name_len);
507:                 zend_call_method_with_1_param(return_value, ce, ce->constructor, "_construct", NULL, &arg1);
508:                 zval_ptr_dtor(&arg1);
509:             } else {
510:                 intern->file_name = estrndup(source->file_name, source->file_name_len);
511:                 intern->file_name_len = source->file_name_len;
512:                 intern->path = spl_filesystem_object_get_path(source, &intern->path_len);
513:                 intern->path = estrndup(intern->path, intern->path_len);
514:             }
515:             break;
516:             case SPL_FS_FILE:
517:                 ce = ce ? ce : source->file_class;
518:
519:                 IF (UNEXPECTED(zend_update_class_constants(ce) != SUCCESS)) {
520:                     break;
521:                 }
522:
523:                 intern = spl_filesystem_from_obj(spl_filesystem_object_new_ex(ce));
524:                 ZVAL_OBJ(return_value, &intern->std);
525:
526:                 spl_filesystem_object_get_file_name(source);
527:
528:                 IF (ce->constructor->common.scope != spl_ce_SplFileInfo) {
529:                     ZVAL_STRING(&arg1, source->file_name, source->file_name_len);
530:                     ZVAL_STRING(&arg2, "r", 1);
531:                     zend_call_method_with_2_param(return_value, ce, ce->constructor, "_construct", NULL, &arg1, &arg2);
532:                     zval_ptr_dtor(&arg1);
533:                     zval_ptr_dtor(&arg2);
534:                 } else {
535:                     intern->file_name = source->file_name;
536:                     intern->file_name_len = source->file_name_len;
537:                     intern->path = spl_filesystem_object_get_path(source, &intern->path_len);
538:                     intern->path = estrndup(intern->path, intern->path_len);
539:
540:                     intern->v.file.open_mode = "r";
541:                     intern->v.file.open_mode_len = 1;
542:
543:                     IF (ht && zend_parse_parameters(ht, "dbr",
544:                         &intern->v.file.open_mode,
545:                         &use_include_path, &intern->v.file.construct == FAILURE)) {
546:                         zend_restore_error_handling(error_handling);
547:                         intern->v.file.open_mode = NULL;
548:                         intern->file_name = NULL;
549:                         zval_ptr_dtor(return_value);
550:                         ZVAL_NULL(return_value);
551:                         return NULL;
552:                     }
553:                 }
554:
555:                 IF (spl_filesystem_file_open(intern, use_include_path, 0) == FAILURE) {
556:                     zend_restore_error_handling(error_handling);
557:                     zval_ptr_dtor(return_value);
558:                     ZVAL_NULL(return_value);
559:                     return NULL;
560:                 }
561:             }
562:         }
563:     }
564: }
565:
566: break;
567: case SPL_FS_DIR:
568:     zend_restore_error_handling(error_handling);
569:     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Operation not supported");
570:     return NULL;
571: }
572: zend_restore_error_handling(error_handling);
573: return NULL;
574: } /* }}} */
575:
576: static int spl_filesystem_is_invalid_or_dot(const char *d_name) /* {{{ */
577: {
578:     IF (d_name[0] == '\0' || spl_filesystem_is_dot(d_name))
579:         return 1;
580:     /* }}} */
581:
582: static char *spl_filesystem_object_get_pathname(spl_filesystem_object *intern, size_t *len) /* {{{ */
583: {
584:     switch (intern->type) {
585:         case SPL_FS_THROW:
586:             case SPL_FS_FILE:
587:                 *len = intern->file_name_len;
588:                 return intern->file_name;
589:             case SPL_FS_DIR:
590:                 IF (intern->v.dir.entry.d_name[0]) {
591:                     spl_filesystem_object_get_file_name(intern);
592:                     *len = intern->file_name_len;
593:                     return intern->file_name;
594:                 }
595:                 *len = 0;
596:                 return NULL;
597:             }
598:     /* }}} */
599:
600: static HashTable *spl_filesystem_object_get_debug_info(zval *object, int *is_temp) /* {{{ */
601: {
602:     spl_filesystem_object *intern = Z_SPL_FILESYSTEM_P(object);
603:     zval tmp;
604:     HashTable *rv;
605:     zend_string *pstr;
606:     char *path;
607:     size_t path_len;
608:     char tmp[2];
609:
610:     *is_temp = 1;
611:
612:     IF (!intern->std.properties) {
613:         rebuild_object_properties(&intern->std);
614:     }
615:     rv = zend_array_dup(intern->std.properties);
616:
617:     pstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "pathName", sizeof("pathName")-1);
618:     path = spl_filesystem_object_get_pathname(intern, &path_len);
619:     ZVAL_STRING(tmp, path, path_len);
620:     zend_symtable_update(rv, pstr, tmp);
621:     zend_string_release(pstr);
622:
623:     IF (intern->file_name) {
624:         pstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "fileName", sizeof("fileName")-1);
625:         spl_filesystem_object_get_path(intern, &path_len);
626:
627:         IF (path_len < path_len + intern->file_name_len) {
628:             ZVAL_STRING(tmp, path, path_len + 1, intern->file_name_len + (path_len + 1));
629:         } else {
630:             ZVAL_STRING(tmp, intern->file_name, intern->file_name_len);
631:         }
632:         zend_symtable_update(rv, pstr, tmp);
633:         zend_string_release(pstr);
634:     }
635:
636:     IF (intern->type == SPL_FS_DIR) {
637:         IF (defined HAVE_GLOB)
638:             pstr = spl_gen_private_prop_name(spl_ce_DirectoryIterator, "glob", sizeof("glob")-1);
639:             IF (spl_gen_stream_is(intern->v.dir.dir, &spl_glob_stream_ops)) {
640:                 ZVAL_STRING(tmp, intern->path, intern->path_len);
641:             } else {
642:                 ZVAL_FALSE(tmp);
643:             }
644:             zend_symtable_update(rv, pstr, tmp);
645:             zend_string_release(pstr);
646:         }
647:         pstr = spl_gen_private_prop_name(spl_ce_RecursiveDirectoryIterator, "subPathName", sizeof("subPathName")-1);
648:         IF (intern->v.dir.sub_path) {
649:             ZVAL_STRING(tmp, intern->v.dir.sub_path, intern->v.dir.sub_path_len);
650:         } else {
651:             ZVAL_EMPTY_STRING(tmp);
652:         }
653:         zend_symtable_update(rv, pstr, tmp);
654:         zend_string_release(pstr);
655:     }
656:
657:     IF (intern->type == SPL_FS_FILE) {
658:         pstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "openMode", sizeof("openMode")-1);
659:         ZVAL_STRING(tmp, intern->v.file.open_mode, intern->v.file.open_mode_len);
660:         zend_symtable_update(rv, pstr, tmp);
661:         zend_string_release(pstr);
662:     }
663:     tmp[0] = '\0';
664:     pstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "delimiter", sizeof("delimiter")-1);
665:     IF (intern->v.file.delimiter) {
666:         ZVAL_STRING(tmp, intern->v.file.delimiter);
667:         zend_symtable_update(rv, pstr, tmp);
668:         zend_string_release(pstr);
669:     }
670:     pstr = spl_gen_private_prop_name(spl_ce_SplFileInfo, "enclosure", sizeof("enclosure")-1);
671:     IF (spl_gen_private_prop_name(spl_ce_SplFileInfo, "enclosure", sizeof("enclosure")-1)) {
672:         ZVAL_STRING(tmp, intern->v.file.enclosure);
673:         zend_symtable_update(rv, pstr, tmp);
674:         zend_string_release(pstr);
675:     }
676:
677:     return rv;
678: } /* }}} */
679:
680: static zend_function *spl_filesystem_object_get_method_check(zend_object **object, zend_string *method, const zval *key) /* {{{ */
681: {
682:     spl_filesystem_object *fsobj = spl_filesystem_from_obj(*object);
683:
684:     IF (fsobj->v.dir.dir == NULL && fsobj->v.dir.path == NULL) {
685:         zend_function *func;
686:         zend_string *tmp = zend_string_init("bad_state_ex", sizeof("bad_state_ex") - 1, 0);
687:         func = zend_get_std_object_handlers()->get_method(object, tmp, NULL);
688:         zend_string_release(tmp);
689:         return func;
690:     }
691:
692:     return zend_get_std_object_handlers()->get_method(object, method, key);
693: } /* }}} */
694:
695: #define DT_CTOR_FLAGS 0x00000001
696: #define DT_CTOR_GLOB 0x00000002
697:
698: void spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAMETERS, zend_long ctor_flags) /* {{{ */
699: {
700:     spl_filesystem_object *intern;
701:     char *path;
702:     int param;
703:     size_t len;
704:     zend_long flags;
705:     zend_error_handling error_handling;
706:
707:     zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, error_handling);
708:
709:     IF (SPL_BAS_FLAG(ctor_flags, DT_CTOR_FLAGS)) {
710:         flags = SPL_FILE_DIR_KEY_AS_PATHNAME|SPL_FILE_DIR_CURRENT_AS_FILEINFO;
711:         parsed = zend_parse_parameters(ZEND_NUM_ARGS(), "s|l", &path, &len, &flags);
712:     } else {
713:         flags = SPL_FILE_DIR_KEY_AS_PATHNAME|SPL_FILE_DIR_CURRENT_AS_SELF;
714:         parsed = zend_parse_parameters(ZEND_NUM_ARGS(), "s", &path, &len);
715:     }
716:
717:     IF (SPL_BAS_FLAG(ctor_flags, SPL_FILE_DIR_SKIPROOTS)) {
718:         flags |= SPL_FILE_DIR_SKIPROOTS;
719:     }
720:     IF (SPL_BAS_FLAG(ctor_flags, SPL_FILE_DIR_UNIXPATHS)) {
721:         flags |= SPL_FILE_DIR_UNIXPATHS;
722:     }
723:     IF (parsed == FAILURE) {
724:         zend_restore_error_handling(error_handling);
725:         return;
726:     }
727:
728:     IF (len) {
729:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Directory name must not be empty.");
730:         zend_restore_error_handling(error_handling);
731:         return;
732:     }
733:
734:     intern->flags = flags;
735:
736:     IF (SPL_BAS_FLAG(ctor_flags, DT_CTOR_GLOB) && strstr(path, "glob://") != path) {
737:         sprintf(&path, "glob://%s", path);
738:     }
739:     spl_filesystem_dir_open(intern, path);
740:     efree(path);
741:
742:     IF (defined HAVE_GLOB)
743:         spl_filesystem_dir_open(intern, path);
744:     }
745:
746:     intern->v.dir.is_recursive = instanceof_function(intern->std.ce, spl_ce_RecursiveDirectoryIterator) ? 1 : 0;
747:     zend_restore_error_handling(error_handling);
748: }

```



```

750: }
751: /* }}} */
752:
753: /* {{{ proto void DirectoryIterator::__construct (string path)
754:  * Constructs a new dir iterator from a path. */
755: SPL_METHOD(DirectoryIterator, __construct)
756: {
757:     spl_filesystem_object_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, 0);
758: }
759: /* }}} */
760:
761: /* {{{ proto void DirectoryIterator::rewind()
762:  * Rewind dir back to the start */
763: SPL_METHOD(DirectoryIterator, rewind)
764: {
765:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
766: }
767: IF (zend_parse_parameters_none() == FAILURE) {
768:     return;
769: }
770:
771: intern->u.dir.index = 0;
772: IF (intern->u.dir.dirp) {
773:     php_stream_rewinddir (intern->u.dir.dirp);
774: }
775: spl_filesystem_dir_read (intern);
776: }
777: /* }}} */
778:
779: /* {{{ proto string DirectoryIterator::key()
780:  * Return current dir entry */
781: SPL_METHOD(DirectoryIterator, key)
782: {
783:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
784: }
785: IF (zend_parse_parameters_none() == FAILURE) {
786:     return;
787: }
788:
789: IF (intern->u.dir.dirp) {
790:     RETURN_LONG (intern->u.dir.index);
791: } else {
792:     RETURN_FALSE;
793: }
794: }
795: /* }}} */
796:
797: /* {{{ proto string DirectoryIterator::current()
798:  * Return this (needed for Iterator interface) */
799: SPL_METHOD(DirectoryIterator, current)
800: {
801:     IF (zend_parse_parameters_none() == FAILURE) {
802:         return;
803:     }
804:     ZVAL_OBJ (&return_value, Z_OBJ_P (getThis()));
805:     Z_ADDREF_P (&return_value);
806: }
807: /* }}} */
808:
809: /* {{{ proto void DirectoryIterator::next()
810:  * Move to next entry */
811: SPL_METHOD(DirectoryIterator, next)
812: {
813:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
814:     int skip_dots = SPL_HAS_FLAG (intern->flags, SPL_FILE_DIR_SKIPDOTS);
815:     IF (zend_parse_parameters_none() == FAILURE) {
816:         return;
817:     }
818: }
819:
820: intern->u.dir.index++;
821: do {
822:     spl_filesystem_dir_read (intern);
823: } while (skip_dots && spl_filesystem_is_dot (intern->u.dir.entry_d_name));
824: IF (intern->u.dir.name) {
825:     /* new (intern->u.dir.name);
826:     intern->u.dir.name = NULL;
827: }
828: }
829: /* }}} */
830:
831: /* {{{ proto void DirectoryIterator::seek (int position)
832:  * Seek to the given position */
833: SPL_METHOD(DirectoryIterator, seek)
834: {
835:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
836:     zval retval;
837:     zend_long pos;
838:     IF (zend_parse_parameters (ZEND_NUM_ARGS(), "l", &pos) == FAILURE) {
839:         return;
840:     }
841: }
842:
843: IF (intern->u.dir.index > pos) {
844:     /* we first rewind */
845:     zend_call_method_with_0_params (&EX (This), Z_OBJCE (EX (This)), &intern->u.dir.func_rewind, "rewind", NULL);
846: }
847:
848: while (intern->u.dir.index < pos) {
849:     int valid = 0;
850:     zend_call_method_with_0_params (&EX (This), Z_OBJCE (EX (This)), &intern->u.dir.func_valid, "valid", &retval);
851:     if (!IS_BOOL (retval)) {
852:         valid = zend_is_true (retval);
853:     }
854:     if (IS_TRUE (retval)) {
855:         zend_throw_exception_ex (spl_ce_OutOfBoundsException, 0, "Seek position '%ZEND_LONG_FMT%' is out of range", pos);
856:     }
857:     return;
858: }
859: zend_call_method_with_0_params (&EX (This), Z_OBJCE (EX (This)), &intern->u.dir.func_next, "next", NULL);
860: }
861: /* }}} */
862:
863: /* {{{ proto string DirectoryIterator::valid()
864:  * Check whether dir contains more entries */
865: SPL_METHOD(DirectoryIterator, valid)
866: {
867:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
868: }
869: IF (zend_parse_parameters_none() == FAILURE) {
870:     return;
871: }
872:
873: RETURN_BOOL (intern->u.dir.entry_d_name[0] != '\0');
874: }
875: /* }}} */
876:
877: /* {{{ proto string SplFileInfo::getPath()
878:  * Return the path */
879: SPL_METHOD(SplFileInfo, getPath)
880: {
881:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
882:     char *path;
883:     size_t path_len;
884:     IF (zend_parse_parameters_none() == FAILURE) {
885:         return;
886:     }
887: }
888:
889: path = spl_filesystem_object_get_path (intern, &path_len);
890: RETURN_STRING (path, path_len);
891: }
892: /* }}} */
893:
894: /* {{{ proto string SplFileInfo::getFilename()
895:  * Return filename only */
896: SPL_METHOD(SplFileInfo, getFilename)
897: {
898:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
899:     size_t path_len;
900:     IF (zend_parse_parameters_none() == FAILURE) {
901:         return;
902:     }
903: }
904:
905: spl_filesystem_object_get_path (intern, &path_len);
906:
907: IF (path_len < path_len - intern->u.dir.name_len) {
908:     RETURN_STRING (intern->u.dir.name + path_len + 1, intern->u.dir.name_len - (path_len + 1));
909: } else {
910:     RETURN_STRING (intern->u.dir.name, intern->u.dir.name_len);
911: }
912: }
913: /* }}} */
914:
915: /* {{{ proto string DirectoryIterator::getFilename()
916:  * Return filename of current dir entry */
917: SPL_METHOD(DirectoryIterator, getFilename)
918: {
919:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
920: }
921: IF (zend_parse_parameters_none() == FAILURE) {
922:     return;
923: }
924:
925: RETURN_STRING (intern->u.dir.entry_d_name);
926: }
927: /* }}} */
928:
929: /* {{{ proto string SplFileInfo::getExtension()
930:  * Return file extension component of path */
931: SPL_METHOD(SplFileInfo, getExtension)
932: {
933:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
934:     char *fname = NULL;
935:     const char *p;
936:     size_t flen;
937:     size_t path_len;

```

```

938:     size_t idx;
939:     zend_string *ret;
940:     IF (zend_parse_parameters_none() == FAILURE) {
941:         return;
942:     }
943: }
944:
945: spl_filesystem_object_get_path (intern, &path_len);
946:
947: IF (path_len < path_len - intern->u.dir.name_len) {
948:     fname = intern->u.dir.name + path_len + 1;
949:     flen = intern->u.dir.name_len - (path_len + 1);
950: } else {
951:     fname = intern->u.dir.name;
952:     flen = intern->u.dir.name_len;
953: }
954:
955: ret = php_basename (fname, flen, NULL, 0);
956:
957: p = zend_search (ZSTR_VAL (ret), '.', ZSTR_LEN (ret));
958: IF (p) {
959:     idx = p - ZSTR_VAL (ret);
960:     RETVAL_STRING (ZSTR_VAL (ret) + idx + 1, ZSTR_LEN (ret) - idx - 1);
961:     zend_string_release (ret);
962:     return;
963: } else {
964:     zend_string_release (ret);
965:     RETURN_EMPTY_STRING ();
966: }
967: }
968: /* }}} */
969:
970: /* {{{ proto string DirectoryIterator::getExtension()
971:  * Return the file extension component of path */
972: SPL_METHOD(DirectoryIterator, getExtension)
973: {
974:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
975:     const char *p;
976:     size_t idx;
977:     zend_string *fname;
978:     IF (zend_parse_parameters_none() == FAILURE) {
979:         return;
980:     }
981: }
982:
983: fname = php_basename (intern->u.dir.entry_d_name, strlen (intern->u.dir.entry_d_name), NULL, 0);
984:
985: p = zend_search (ZSTR_VAL (fname), '.', ZSTR_LEN (fname));
986: IF (p) {
987:     idx = p - ZSTR_VAL (fname);
988:     RETVAL_STRING (ZSTR_VAL (fname) + idx + 1, ZSTR_LEN (fname) - idx - 1);
989:     zend_string_release (fname);
990:     return;
991: } else {
992:     zend_string_release (fname);
993:     RETURN_EMPTY_STRING ();
994: }
995: /* }}} */
996:
997: /* {{{ proto string SplFileInfo::getBaseName (string suffix)
998:  * Return filename component of path */
999: SPL_METHOD(SplFileInfo, getBaseName)
1000: {
1001:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
1002:     char *fname, *suffix = 0;
1003:     size_t flen;
1004:     size_t slen = 0, path_len;
1005:     IF (zend_parse_parameters (ZEND_NUM_ARGS(), "s", &suffix, &slen) == FAILURE) {
1006:         return;
1007:     }
1008: }
1009:
1010: spl_filesystem_object_get_path (intern, &path_len);
1011:
1012: IF (path_len < path_len - intern->u.dir.name_len) {
1013:     fname = intern->u.dir.name + path_len + 1;
1014:     flen = intern->u.dir.name_len - (path_len + 1);
1015: } else {
1016:     fname = intern->u.dir.name;
1017:     flen = intern->u.dir.name_len;
1018: }
1019:
1020: RETURN_STR (php_basename (fname, flen, suffix, slen));
1021: }
1022: /* }}} */
1023:
1024: /* {{{ proto string DirectoryIterator::getBaseName (string suffix)
1025:  * Return filename component of current dir entry */
1026: SPL_METHOD(DirectoryIterator, getBaseName)
1027: {
1028:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
1029:     char *suffix = 0;
1030:     size_t slen = 0;
1031:     zend_string *fname;
1032:     IF (zend_parse_parameters (ZEND_NUM_ARGS(), "s", &suffix, &slen) == FAILURE) {
1033:         return;
1034:     }
1035: }
1036:
1037: fname = php_basename (intern->u.dir.entry_d_name, strlen (intern->u.dir.entry_d_name), suffix, slen);
1038:
1039: RETVAL_STR (fname);
1040: }
1041: /* }}} */
1042:
1043: /* {{{ proto string SplFileInfo::getPathname()
1044:  * Return path and filename */
1045: SPL_METHOD(SplFileInfo, getPathname)
1046: {
1047:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
1048:     char *path;
1049:     size_t path_len;
1050:     IF (zend_parse_parameters_none() == FAILURE) {
1051:         return;
1052:     }
1053: }
1054:
1055: path = spl_filesystem_object_get_pathname (intern, &path_len);
1056: IF (path != NULL) {
1057:     RETURN_STRING (path, path_len);
1058: } else {
1059:     RETURN_FALSE;
1060: }
1061: /* }}} */
1062:
1063: /* {{{ proto string FilesystemIterator::key()
1064:  * Return getFilename() or getFileInfo() depending on flags */
1065: SPL_METHOD(FilesystemIterator, key)
1066: {
1067:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
1068: }
1069: IF (zend_parse_parameters_none() == FAILURE) {
1070:     return;
1071: }
1072:
1073: IF (SPL_FILE_DIR_CURRENT (intern, SPL_FILE_DIR_CURRENT_AS_PATHNAME)) {
1074:     RETURN_STRING (intern->u.dir.entry_d_name);
1075: } else {
1076:     spl_filesystem_object_get_file_name (intern);
1077:     RETURN_STRING (intern->u.dir.name, intern->u.dir.name_len);
1078: }
1079: }
1080: /* }}} */
1081:
1082: /* {{{ proto string FilesystemIterator::current()
1083:  * Return getFilename() or getFileInfo() depending on flags */
1084: SPL_METHOD(FilesystemIterator, current)
1085: {
1086:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
1087: }
1088: IF (zend_parse_parameters_none() == FAILURE) {
1089:     return;
1090: }
1091:
1092: IF (SPL_FILE_DIR_CURRENT (intern, SPL_FILE_DIR_CURRENT_AS_PATHNAME)) {
1093:     spl_filesystem_object_get_file_name (intern);
1094:     RETURN_STRING (intern->u.dir.name, intern->u.dir.name_len);
1095: } else IF (SPL_FILE_DIR_CURRENT (intern, SPL_FILE_DIR_CURRENT_AS_FILEINFO)) {
1096:     spl_filesystem_object_get_file_name (intern);
1097:     spl_filesystem_object_get_cwata_type (0, intern, SPL_FS_INFO, NULL, &return_value);
1098: } else {
1099:     ZVAL_OBJ (&return_value, Z_OBJ_P (getThis()));
1100:     Z_ADDREF_P (&return_value);
1101:     /* RETURN_STRING (intern->u.dir.entry_d_name, 1); */
1102: }
1103: }
1104: /* }}} */
1105:
1106: /* {{{ proto bool DirectoryIterator::isDot()
1107:  * Return true if current entry is '.' or '..' */
1108: SPL_METHOD(DirectoryIterator, isDot)
1109: {
1110:     spl_filesystem_object_construct ('intern = Z_SPFILESYSTEM_P (getThis());
1111: }
1112: IF (zend_parse_parameters_none() == FAILURE) {
1113:     return;
1114: }
1115:
1116: RETURN_BOOL (spl_filesystem_is_dot (intern->u.dir.entry_d_name));
1117: }
1118: /* }}} */
1119:
1120: /* {{{ proto void SplFileInfo::__construct (string filename)
1121:  * Constructs a new SplFileInfo from a path */
1122: /* When the constructor gets called the object is already created
1123:  * by the engine, so we must only call 'additional' initializations.
1124:  */
1125: SPL_METHOD(SplFileInfo, __construct)

```

```

1126: {
1127:     spl_filesystem_object *intern;
1128:     char *path;
1129:     size_t len;
1130:
1131:     if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "s", &path, &len) == FAILURE) {
1132:         return;
1133:     }
1134:
1135:     intern = _spl_filesystem_P(getThis());
1136:
1137:     spl_filesystem_info_set_filename(intern, path, len, 1);
1138:
1139:     /* intern->type = SPL_FS_INFO already set */
1140:
1141:     /* {{{ */
1142:
1143:     /* {{{ FileinfoFunction */
1144:     FileinfoFunction(func_name, func_num) {
1145:         SPL_METHOD(splFileInfo, func_name) {
1146:             \
1147:             spl_filesystem_object *intern = _spl_filesystem_P(getThis()); \
1148:             zend_error_handling error_handling; \
1149:             if (zend_parse_parameters_none() == FAILURE) { \
1150:                 return; \
1151:             } \
1152:             \
1153:             zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling); \
1154:             spl_filesystem_object_get_file_name(intern); \
1155:             php_stat(intern->file_name, intern->file_name_len, func_num, return_value); \
1156:             zend_restore_error_handling(&error_handling); \
1157:             \
1158:             /* }}} */
1159:
1160:     /* {{{ proto int splFileInfo::getPerms()
1161:      * Get file permissions */
1162:     FileinfoFunction(getPerms, FS_PERMS)
1163:     /* }}} */
1164:
1165:     /* {{{ proto int splFileInfo::getInode()
1166:      * Get file inode */
1167:     FileinfoFunction(getInode, FS_INODE)
1168:     /* }}} */
1169:
1170:     /* {{{ proto int splFileInfo::getSize()
1171:      * Get file size */
1172:     FileinfoFunction(getSize, FS_SIZE)
1173:     /* }}} */
1174:
1175:     /* {{{ proto int splFileInfo::getOwner()
1176:      * Get file owner */
1177:     FileinfoFunction(getOwner, FS_OWNER)
1178:     /* }}} */
1179:
1180:     /* {{{ proto int splFileInfo::getGroup()
1181:      * Get file group */
1182:     FileinfoFunction(getGroup, FS_GROUP)
1183:     /* }}} */
1184:
1185:     /* {{{ proto int splFileInfo::getTime()
1186:      * Get last access time of file */
1187:     FileinfoFunction(getAtime, FS_ATIME)
1188:     /* }}} */
1189:
1190:     /* {{{ proto int splFileInfo::getMTime()
1191:      * Get last modification time of file */
1192:     FileinfoFunction(getMTime, FS_MTIME)
1193:     /* }}} */
1194:
1195:     /* {{{ proto int splFileInfo::getCTime()
1196:      * Get inode modification time of file */
1197:     FileinfoFunction(getCtime, FS_CTIME)
1198:     /* }}} */
1199:
1200:     /* {{{ proto string splFileInfo::getType()
1201:      * Get file type */
1202:     FileinfoFunction(getType, FS_TYPE)
1203:     /* }}} */
1204:
1205:     /* {{{ proto bool splFileInfo::isWritable()
1206:      * Returns true if file can be written */
1207:     FileinfoFunction(isWritable, FS_IS_W)
1208:     /* }}} */
1209:
1210:     /* {{{ proto bool splFileInfo::isReadable()
1211:      * Returns true if file can be read */
1212:     FileinfoFunction(isReadable, FS_IS_R)
1213:     /* }}} */
1214:
1215:     /* {{{ proto bool splFileInfo::isExecutable()
1216:      * Returns true if file is executable */
1217:     FileinfoFunction(isExecutable, FS_IS_X)
1218:     /* }}} */
1219:
1220:     /* {{{ proto bool splFileInfo::isFile()
1221:      * Returns true if file is a regular file */
1222:     FileinfoFunction(isFile, FS_IS_FILE)
1223:     /* }}} */
1224:
1225:     /* {{{ proto bool splFileInfo::isDir()
1226:      * Returns true if file is directory */
1227:     FileinfoFunction(isDir, FS_IS_DIR)
1228:     /* }}} */
1229:
1230:     /* {{{ proto bool splFileInfo::isLink()
1231:      * Returns true if file is symbolic link */
1232:     FileinfoFunction(isLink, FS_IS_LINK)
1233:     /* }}} */
1234:
1235:     /* {{{ proto string splFileInfo::getLinkTarget()
1236:      * Return the target of a symbolic link */
1237:     SPL_METHOD(splFileInfo, getLinkTarget)
1238:     /* }}} */
1239:
1240:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1241:     symlink &ret;
1242:     char buff[MAXPATHLEN];
1243:     zend_error_handling error_handling;
1244:
1245:     if (zend_parse_parameters_none() == FAILURE) {
1246:         return;
1247:     }
1248:
1249:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
1250:
1251:     if (defined(PHP_WIN32)) { HAVE_SYMLINK
1252:     if (intern->file_name == NULL) {
1253:         php_error_docref(NULL, E_WARNING, "Empty filename");
1254:         return FALSE;
1255:     } else if (!IS_ABSOLUTE_PATH(intern->file_name, intern->file_name_len)) {
1256:         char expanded_path[MAXPATHLEN];
1257:         if (expand_filepath_with_mode(intern->file_name, expanded_path, NULL, 0, CHD_EXPAND)) {
1258:             php_error_docref(NULL, E_WARNING, "No such file or directory");
1259:             return FALSE;
1260:         }
1261:         ret = php_sya_readlink(expanded_path, buff, MAXPATHLEN - 1);
1262:     } else {
1263:         ret = php_sya_readlink(intern->file_name, buff, MAXPATHLEN - 1);
1264:     }
1265:     if (ret == -1) /* always fail if not implemented */
1266:         return;
1267:
1268:     if (ret == -1) {
1269:         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Unable to read link %s, error: %s", intern->file_name, strerror(errno));
1270:         return FALSE;
1271:     } else {
1272:         /* Append NULL to the end of the string */
1273:         buff[ret] = '\0';
1274:
1275:         RETVAL_STRING(buff, ret);
1276:     }
1277:
1278:     zend_restore_error_handling(&error_handling);
1279:     /* }}} */
1280:     /* }}} */
1281:
1282:     if (HAVE_REALPATH) {
1283:     if (defined(PHP_WIN32)) {
1284:     /* {{{ proto string splFileInfo::getRealPath()
1285:      * Return the resolved path */
1286:     SPL_METHOD(splFileInfo, getRealPath)
1287:     /* }}} */
1288:
1289:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1290:     char buff[MAXPATHLEN];
1291:     char *filename;
1292:     zend_error_handling error_handling;
1293:
1294:     if (zend_parse_parameters_none() == FAILURE) {
1295:         return;
1296:     }
1297:
1298:     zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
1299:
1300:     if (intern->type == SPL_FS_DIR && !intern->file_name && intern->u.dir.entry_dname[0]) {
1301:         spl_filesystem_object_get_file_name(intern);
1302:     }
1303:
1304:     if (intern->orig_path) {
1305:         filename = intern->orig_path;
1306:     } else {
1307:         filename = intern->file_name;
1308:     }
1309:
1310:     if (filename && !VOWD_REALPATH(filename, buff)) {
1311:         if (VOWD_ACCESS(buff, F_OK)) {
1312:             RETVAL_FALSE;
1313:         } else

```

```

1314:         return;
1315:     }
1316:     } else {
1317:         RETVAL_FALSE;
1318:     }
1319:
1320:     zend_restore_error_handling(&error_handling);
1321:     /* }}} */
1322:     /* }}} */
1323:
1324:     /* {{{ proto void splFileInfo::openFile(string mode = "r", bool use_include_path = false, resource context)
1325:      * Open the current file */
1326:     SPL_METHOD(splFileInfo, openFile)
1327:     /* }}} */
1328:     /* }}} */
1329:
1330:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1331:
1332:     spl_filesystem_object_create_type(ZEND_NUM_ARGS(), intern, SPL_FS_FILE, NULL, return_value);
1333:     /* }}} */
1334:
1335:     /* {{{ proto void splFileInfo::setFileClass(string class_name)
1336:      * Class to use in openFile() */
1337:     SPL_METHOD(splFileInfo, setFileClass)
1338:     /* }}} */
1339:
1340:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1341:     zend_class_entry *ce = spl_ce_splFileInfo;
1342:     zend_error_handling error_handling;
1343:
1344:     zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);
1345:
1346:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
1347:         intern->file_class = ce;
1348:     }
1349:
1350:     zend_restore_error_handling(&error_handling);
1351:     /* }}} */
1352:
1353:     /* {{{ proto void splFileInfo::setPathInfo(string class_name)
1354:      * Class to use in getPathInfo(), getFileInfo() */
1355:     SPL_METHOD(splFileInfo, setPathInfoClass)
1356:     /* }}} */
1357:
1358:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1359:     zend_class_entry *ce = spl_ce_splFileInfo;
1360:     zend_error_handling error_handling;
1361:
1362:     zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);
1363:
1364:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
1365:         intern->path_info_class = ce;
1366:     }
1367:
1368:     zend_restore_error_handling(&error_handling);
1369:     /* }}} */
1370:
1371:     /* {{{ proto splFileInfo splFileInfo::getFileInfo(string class_name)
1372:      * Get/copy file info */
1373:     SPL_METHOD(splFileInfo, getFileInfo)
1374:     /* }}} */
1375:
1376:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1377:     zend_class_entry *ce = intern->path_info_class;
1378:     zend_error_handling error_handling;
1379:
1380:     zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);
1381:
1382:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
1383:         spl_filesystem_object_create_type(ZEND_NUM_ARGS(), intern, SPL_FS_INFO, ce, return_value);
1384:     }
1385:
1386:     zend_restore_error_handling(&error_handling);
1387:     /* }}} */
1388:
1389:     /* {{{ proto splFileInfo splFileInfo::getPathInfo(string class_name)
1390:      * Get/copy file info */
1391:     SPL_METHOD(splFileInfo, getPathInfo)
1392:     /* }}} */
1393:
1394:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1395:     zend_class_entry *ce = intern->path_info_class;
1396:     zend_error_handling error_handling;
1397:
1398:     zend_replace_error_handling(EH_THROW, spl_ce_UnexpectedValueException, &error_handling);
1399:
1400:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "|C", &ce) == SUCCESS) {
1401:         char *path = spl_filesystem_object_get_pathname(intern, &path_len);
1402:         if (path) {
1403:             char *dpath = estrndup(path, path_len);
1404:             path_len = php_dirname(dpath, path_len);
1405:             spl_filesystem_object_create_info(intern, dpath, path_len, 1, ce, return_value);
1406:             efree(dpath);
1407:         }
1408:     }
1409:
1410:     zend_restore_error_handling(&error_handling);
1411:     /* }}} */
1412:     /* }}} */
1413:
1414:     /* {{{ proto void splFileInfo::bad_state_ex(void)
1415:      * splFileInfo::bad_state_ex()
1416:      *
1417:      * The parent constructor was not called: the object is in an "
1418:      * invalid state" */
1419:     /* }}} */
1420:
1421:     /* {{{ proto void FilesystemIterator::__construct(string path, int flags)
1422:      * Constructs a new dir iterator from a path */
1423:     SPL_METHOD(FilesystemIterator, __construct)
1424:     /* }}} */
1425:
1426:     spl_filesystem_object_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, DIT_CTOR_FLAGS | SPL_FILE_DIR_SKIPDOTS);
1427:
1428:     /* }}} */
1429:     /* }}} */
1430:
1431:     /* {{{ proto void FilesystemIterator::rewind()
1432:      * Rewind dir back to the start */
1433:     SPL_METHOD(FilesystemIterator, rewind)
1434:     /* }}} */
1435:
1436:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1437:     int skip_dots = SPL_HAS_FLAG(intern->flags, SPL_FILE_DIR_SKIPDOTS);
1438:
1439:     if (zend_parse_parameters_none() == FAILURE) {
1440:         return;
1441:     }
1442:
1443:     intern->u.dir.index = 0;
1444:     if (intern->u.dir.dirp) {
1445:         php_stream_rewinddir(intern->u.dir.dirp);
1446:     }
1447:
1448:     do {
1449:         spl_filesystem_dir_read(intern);
1450:     } while (skip_dots && spl_filesystem_is_dot(intern->u.dir.entry_dname));
1451:
1452:     /* }}} */
1453:
1454:     /* {{{ proto int FilesystemIterator::getFlags()
1455:      * Get handling flags */
1456:     SPL_METHOD(FilesystemIterator, getFlags)
1457:     /* }}} */
1458:
1459:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1460:     zend_long flags;
1461:
1462:     if (zend_parse_parameters_none() == FAILURE) {
1463:         return;
1464:     }
1465:
1466:     RETURN_LONG(intern->flags & (SPL_FILE_DIR_KEY_MODE_MASK | SPL_FILE_DIR_CURRENT_MODE_MASK | SPL_FILE_DIR_OTHERS_MASK));
1467:     /* }}} */
1468:
1469:     /* {{{ proto void FilesystemIterator::setFlags(long $flags)
1470:      * Set handling flags */
1471:     SPL_METHOD(FilesystemIterator, setFlags)
1472:     /* }}} */
1473:
1474:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1475:     zend_long flags;
1476:
1477:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &flags) == FAILURE) {
1478:         return;
1479:     }
1480:
1481:     RETURN_LONG(intern->flags & (SPL_FILE_DIR_KEY_MODE_MASK | SPL_FILE_DIR_CURRENT_MODE_MASK | SPL_FILE_DIR_OTHERS_MASK) | flags);
1482:     /* }}} */
1483:
1484:     /* {{{ proto bool RecursiveDirectoryIterator::hasChildren(bool $allow_links = false)
1485:      * Returns whether current entry is a directory and not "." or ".." */
1486:     SPL_METHOD(RecursiveDirectoryIterator, hasChildren)
1487:     /* }}} */
1488:
1489:     zend_bool allow_links = 0;
1490:     spl_filesystem_object *intern = _spl_filesystem_P(getThis());
1491:
1492:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "|b", &allow_links) == FAILURE) {
1493:         return;
1494:     }
1495:
1496:     if (spl_filesystem_is_invalid_or_dot(intern->u.dir.entry_dname)) {
1497:         return FALSE;
1498:     } else {
1499:         spl_filesystem_object_get_file_name(intern);
1500:         if (allow_links && (intern->flags & SPL_FILE_DIR_FOLLOW_SYMLINKS)) {
1501:             php_stat(intern->file_name, intern->file_name_len, FS_IS_LINK, return_value);
1502:             if (zend_is_true(return_value)) {
1503:                 return TRUE;
1504:             }
1505:         }
1506:     }
1507:
1508:     php_stat(intern->file_name, intern->file_name_len, FS_IS_DIR, return_value);
1509:     /* }}} */

```

```

1690: /* {{{ spl_filesystem_dir_it_current_key */
1691: static void spl_filesystem_dir_it_current_key(zend_object_iterator *iter, zval *key)
1692: {
1693:     spl_filesystem_object *object = spl_filesystem_iterator_to_object(spl_filesystem_iterator *iter);
1694:
1695:     ZVAL_STRING(key, object->u.dir.index);
1696: }
1697:
1698: /* }}} */
1699:
1700: /* {{{ spl_filesystem_dir_it_move_forward */
1701: static void spl_filesystem_dir_it_move_forward(zend_object_iterator *iter)
1702: {
1703:     spl_filesystem_object *object = spl_filesystem_iterator_to_object(spl_filesystem_iterator *iter);
1704:
1705:     object->u.dir.index++;
1706:     spl_filesystem_dir_read(object);
1707:     if (object->u.dir.name) {
1708:         zfree(object->u.dir.name);
1709:         object->u.dir.name = NULL;
1710:     }
1711: }
1712: /* }}} */
1713:
1714: /* {{{ spl_filesystem_dir_it_rewind */
1715: static void spl_filesystem_dir_it_rewind(zend_object_iterator *iter)
1716: {
1717:     spl_filesystem_object *object = spl_filesystem_iterator_to_object(spl_filesystem_iterator *iter);
1718:
1719:     object->u.dir.index = 0;
1720:     if (object->u.dir.dirp) {
1721:         php_stream_rewinddir(object->u.dir.dirp);
1722:     }
1723:     spl_filesystem_dir_read(object);
1724: }
1725: /* }}} */
1726:
1727: /* {{{ spl_filesystem_tree_it_dtor */
1728: static void spl_filesystem_tree_it_dtor(zend_object_iterator *iter)
1729: {
1730:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1731:
1732:     if (!IS_ITERATOR(iterator->intern.data)) {
1733:         zval *object = iterator->intern.data;
1734:         zval_ptr_dtor(object);
1735:     } else {
1736:         if (!IS_ITERATOR(iterator->current)) {
1737:             zval_ptr_dtor(iterator->current);
1738:             ZVAL_UNDEF(iterator->current);
1739:         }
1740:     }
1741: }
1742: /* }}} */
1743:
1744: /* {{{ spl_filesystem_tree_if_current_data */
1745: static void *spl_filesystem_tree_if_current_data(zend_object_iterator *iter)
1746: {
1747:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1748:     spl_filesystem_object *object = spl_filesystem_iterator_to_object(iterator);
1749:
1750:     if (SPL_FILE_DIR_CURRENT(object, SPL_FILE_DIR_CURRENT_AS_PATHNAME)) {
1751:         if (!IS_ITERATOR(iterator->current)) {
1752:             spl_filesystem_object_get_file_name(object);
1753:             ZVAL_STRING(iterator->current, object->u.file_name, object->u.file_name_len);
1754:         }
1755:         return iterator->current;
1756:     } else if (SPL_FILE_DIR_CURRENT(object, SPL_FILE_DIR_CURRENT_AS_FILEINFO)) {
1757:         if (!IS_ITERATOR(iterator->current)) {
1758:             spl_filesystem_object_get_file_name(object);
1759:             spl_filesystem_object_create_type(0, object, SPL_FI_INFO, NULL, iterator->current);
1760:         }
1761:         return iterator->current;
1762:     } else {
1763:         return iterator->intern.data;
1764:     }
1765: }
1766: /* }}} */
1767:
1768: /* {{{ spl_filesystem_tree_if_current_key */
1769: static void spl_filesystem_tree_if_current_key(zend_object_iterator *iter, zval *key)
1770: {
1771:     spl_filesystem_object *object = spl_filesystem_iterator_to_object(spl_filesystem_iterator *iter);
1772:
1773:     if (SPL_FILE_DIR_KEY(object, SPL_FILE_DIR_KEY_AS_FILENAME)) {
1774:         ZVAL_STRING(key, object->u.dir.entry_d_name);
1775:     } else {
1776:         spl_filesystem_object_get_file_name(object);
1777:         ZVAL_STRING(key, object->u.file_name, object->u.file_name_len);
1778:     }
1779: }
1780: /* }}} */
1781:
1782: /* {{{ spl_filesystem_tree_if_move_forward */
1783: static void spl_filesystem_tree_if_move_forward(zend_object_iterator *iter)
1784: {
1785:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1786:     spl_filesystem_object *object = spl_filesystem_iterator_to_object(iterator);
1787:
1788:     object->u.dir.index++;
1789:     do {
1790:         spl_filesystem_dir_read(object);
1791:     } while (spl_filesystem_is_dot(object->u.dir.entry_d_name));
1792:     if (object->u.dir.name) {
1793:         zfree(object->u.dir.name);
1794:         object->u.dir.name = NULL;
1795:     }
1796:     if (!IS_ITERATOR(iterator->current)) {
1797:         zval_ptr_dtor(iterator->current);
1798:         ZVAL_UNDEF(iterator->current);
1799:     }
1800: }
1801: /* }}} */
1802:
1803: /* {{{ spl_filesystem_tree_it_rewind */
1804: static void spl_filesystem_tree_it_rewind(zend_object_iterator *iter)
1805: {
1806:     spl_filesystem_iterator *iterator = (spl_filesystem_iterator *)iter;
1807:     spl_filesystem_object *object = spl_filesystem_iterator_to_object(iterator);
1808:
1809:     object->u.dir.index = 0;
1810:     if (object->u.dir.dirp) {
1811:         php_stream_rewinddir(object->u.dir.dirp);
1812:     }
1813:     do {
1814:         spl_filesystem_dir_read(object);
1815:     } while (spl_filesystem_is_dot(object->u.dir.entry_d_name));
1816:     if (!IS_ITERATOR(iterator->current)) {
1817:         zval_ptr_dtor(iterator->current);
1818:         ZVAL_UNDEF(iterator->current);
1819:     }
1820: }
1821: /* }}} */
1822:
1823: /* {{{ iterator handler table */
1824: static const zend_object_iterator_funcs spl_filesystem_tree_if_funcs = {
1825:     spl_filesystem_tree_it_dtor,
1826:     spl_filesystem_dir_it_valid,
1827:     spl_filesystem_tree_if_current_data,
1828:     spl_filesystem_tree_if_current_key,
1829:     spl_filesystem_tree_if_move_forward,
1830:     spl_filesystem_tree_it_rewind,
1831:     NULL
1832: };
1833: /* }}} */
1834:
1835: /* {{{ spl_dir_dir_iterator */
1836: zend_object_iterator *spl_filesystem_tree_get_iterator(zend_class_entry *ce, zval *object, int by_ref)
1837: {
1838:     spl_filesystem_iterator *iterator;
1839:     spl_filesystem_object *uio;
1840:
1841:     if (by_ref) {
1842:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1843:         return NULL;
1844:     }
1845:     dir_object = Z_SPLFILESYSTEM_P(object);
1846:     iterator = spl_filesystem_object_to_iterator(dir_object);
1847:
1848:     ZVAL_COPY(iterator->intern.data, object);
1849:     iterator->intern.funcs = spl_filesystem_tree_if_funcs;
1850:
1851:     return iterator->intern;
1852: }
1853: /* }}} */
1854:
1855: /* {{{ spl_filesystem_object_cast */
1856: static int spl_filesystem_object_cast(zval *readobj, zval *writeobj, int type)
1857: {
1858:     spl_filesystem_object *intern = Z_SPLFILESYSTEM_P(readobj);
1859:
1860:     if (type == IS_STRING) {
1861:         if (!IS_OBJECT_P(readobj))>to_string();
1862:         return std_object_handlers->cast_object(readobj, writeobj, type);
1863:     }
1864:
1865:     switch (intern->type) {
1866:     case SPL_FI_INFO:
1867:     case SPL_FI_FILE:
1868:         ZVAL_STRING(writeobj, intern->u.file_name, intern->u.file_name_len);
1869:         return SUCCESS;
1870:     case SPL_FI_DIR:
1871:         ZVAL_STRING(writeobj, intern->u.dir.entry_d_name);
1872:         return SUCCESS;
1873:     }
1874:
1875:     else if (type == IS_BOOL) {
1876:         ZVAL_TRUE(writeobj);
1877:         return SUCCESS;
1878:     }

```

```

1870: return FAILURE;
1880: }
1890: }
1900: }
1910: }
1920: }
1930: }
1940: }
1950: }
1960: }
1970: }
1980: }
1990: }
2000: }
2010: }
2020: }
2030: }
2040: }
2050: }
2060: }
2070: }
2080: }
2090: }
2100: }
2110: }
2120: }
2130: }
2140: }
2150: }
2160: }
2170: }
2180: }
2190: }
2200: }
2210: }
2220: }
2230: }
2240: }
2250: }
2260: }
2270: }
2280: }
2290: }
2300: }
2310: }
2320: }
2330: }
2340: }
2350: }
2360: }
2370: }
2380: }
2390: }
2400: }
2410: }
2420: }
2430: }
2440: }
2450: }
2460: }
2470: }
2480: }
2490: }
2500: }
2510: }
2520: }
2530: }
2540: }
2550: }
2560: }
2570: }
2580: }
2590: }
2600: }
2610: }
2620: }
2630: }
2640: }
2650: }
2660: }
2670: }
2680: }
2690: }
2700: }
2710: }
2720: }
2730: }
2740: }
2750: }
2760: }
2770: }
2780: }
2790: }
2800: }
2810: }
2820: }
2830: }
2840: }
2850: }
2860: }
2870: }
2880: }
2890: }
2900: }
2910: }
2920: }
2930: }
2940: }
2950: }
2960: }
2970: }
2980: }
2990: }
3000: }
3010: }
3020: }
3030: }
3040: }
3050: }
3060: }
3070: }
3080: }
3090: }
3100: }
3110: }
3120: }
3130: }
3140: }
3150: }
3160: }
3170: }
3180: }
3190: }
3200: }
3210: }
3220: }
3230: }
3240: }
3250: }
3260: }
3270: }
3280: }
3290: }
3300: }
3310: }
3320: }
3330: }
3340: }
3350: }
3360: }
3370: }
3380: }
3390: }
3400: }
3410: }
3420: }
3430: }
3440: }
3450: }
3460: }
3470: }
3480: }
3490: }
3500: }
3510: }
3520: }
3530: }
3540: }
3550: }
3560: }
3570: }
3580: }
3590: }
3600: }
3610: }
3620: }
3630: }
3640: }
3650: }
3660: }
3670: }
3680: }
3690: }
3700: }
3710: }
3720: }
3730: }
3740: }
3750: }
3760: }
3770: }
3780: }
3790: }
3800: }
3810: }
3820: }
3830: }
3840: }
3850: }
3860: }
3870: }
3880: }
3890: }
3900: }
3910: }
3920: }
3930: }
3940: }
3950: }
3960: }
3970: }
3980: }
3990: }
4000: }

```

```

2253: spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2254: zend_bool use_include_path = 0;
2255: char *pl, *pz;
2256: char *tmp_path;
2257: size_t tmp_path_len;
2258: zend_error_handling error_handling;
2259:
2260: intern->u.file.open_mode = NULL;
2261: intern->u.file.open_mode_len = 0;
2262:
2263: IF (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "p|abst",
2264:     &intern->file_name, &intern->file_name_len,
2265:     &intern->u.file.open_mode, &intern->u.file.open_mode_len,
2266:     &use_include_path, &intern->u.file.sconnext) == FAILURE) {
2267:     intern->u.file.open_mode = NULL;
2268:     &intern->file_name = NULL;
2269:     return;
2270: }
2271:
2272: IF (intern->u.file.open_mode == NULL) {
2273:     intern->u.file.open_mode = "r";
2274:     intern->u.file.open_mode_len = 1;
2275: }
2276:
2277: zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
2278:
2279: IF (spl_filesystem_file_open(intern, use_include_path, 0) == SUCCESS) {
2280:     tmp_path_len = strlen(intern->u.file.stream->orig_path);
2281:
2282:     IF (tmp_path_len > 1 && IS_SLASH_AT(intern->u.file.stream->orig_path, tmp_path_len-1)) {
2283:         tmp_path_len--;
2284:     }
2285:
2286:     tmp_path = estrndup(intern->u.file.stream->orig_path, tmp_path_len);
2287:
2288:     p1 = strchr(tmp_path, '/');
2289:     IF defined(PHP_WIN32)
2290:         p2 = strchr(tmp_path, '\\');
2291:     while
2292:         p2 < p1 {
2293:             IF (p1 || p2) {
2294:                 intern->u_path_len = ((p1 > p2 ? p1 : p2) - tmp_path);
2295:             } ELSE {
2296:                 intern->u_path_len = 0;
2297:             }
2298:             ofree(tmp_path);
2299:
2300:             intern->u_path = estrndup(intern->u.file.stream->orig_path, intern->u_path_len);
2301:         }
2302:
2303:     zend_restore_error_handling(&error_handling);
2304:
2305:     /* }}} */
2306:
2307:     /* {{{ proto void SplFileObject::__construct([int max_memory])
2308:     Construct a new temp file object */
2309:     SPL_METHOD(SplFileObject, __construct)
2310:     {
2311:         zend_long max_memory = PHP_STREAM_MAX_LEN;
2312:         char tmp_fname[64];
2313:         spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2314:         zend_error_handling error_handling;
2315:
2316:         IF (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "|l", &max_memory) == FAILURE) {
2317:             return;
2318:         }
2319:
2320:         IF (max_memory < 0) {
2321:             intern->file_name = "php://memory";
2322:             intern->file_name_len = 12;
2323:         } ELSE IF (ZEND_NUM_ARGS() > 1) {
2324:             intern->file_name_len = sprintf(tmp_fname, sizeof(tmp_fname), "php://temp/memmemory:" ZEND_LONG_FMT, max_memory);
2325:             intern->file_name = tmp_fname;
2326:         } ELSE {
2327:             intern->file_name = "php://temp";
2328:             intern->file_name_len = 10;
2329:         }
2330:
2331:         intern->u.file.open_mode = "wb";
2332:         intern->u.file.open_mode_len = 1;
2333:
2334:         zend_replace_error_handling(EH_THROW, spl_ce_RuntimeException, &error_handling);
2335:         IF (spl_filesystem_file_open(intern, 0, 0) == SUCCESS) {
2336:             intern->u_path_len = 0;
2337:             intern->u_path = estrndup("", 0);
2338:         }
2339:
2340:         zend_restore_error_handling(&error_handling);
2341:     } /* }}} */
2342:
2343:     /* {{{ proto void SplFileObject::rewind()
2344:     Rewind the file and read the first line */
2345:     SPL_METHOD(SplFileObject, rewind)
2346:     {
2347:         spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2348:
2349:         IF (zend_parse_parameters_none() == FAILURE) {
2350:             return;
2351:         }
2352:
2353:         spl_filesystem_file_rewind(getThis(), intern);
2354:     } /* }}} */
2355:
2356:     /* {{{ proto void SplFileObject::eof()
2357:     Return whether end of file is reached */
2358:     SPL_METHOD(SplFileObject, eof)
2359:     {
2360:         spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2361:
2362:         IF (zend_parse_parameters_none() == FAILURE) {
2363:             return;
2364:         }
2365:
2366:         IF(!intern->u.file.stream) {
2367:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2368:             return;
2369:         }
2370:
2371:         RETURN_BOOL(pphp_stream_eof(intern->u.file.stream));
2372:     } /* }}} */
2373:
2374:     /* {{{ proto void SplFileObject::valid()
2375:     Return true if */
2376:     SPL_METHOD(SplFileObject, valid)
2377:     {
2378:         spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2379:
2380:         IF (zend_parse_parameters_none() == FAILURE) {
2381:             return;
2382:         }
2383:
2384:         IF (spl_was_flag(intern->flags, SPL_FILE_OBJECT_READ Ahead)) {
2385:             RETURN_BOOL(intern->u.file.current_line || !IS_HUNDERF(intern->u.file.current_val));
2386:         } ELSE {
2387:             IF(!intern->u.file.stream) {
2388:                 RETURN_FALSE;
2389:             }
2390:             RETURN_BOOL(pphp_stream_eof(intern->u.file.stream));
2391:         }
2392:     } /* }}} */
2393:
2394:     /* {{{ proto string SplFileObject::fgets()
2395:     Return next line from file */
2396:     SPL_METHOD(SplFileObject, fgets)
2397:     {
2398:         spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2399:
2400:         IF (zend_parse_parameters_none() == FAILURE) {
2401:             return;
2402:         }
2403:
2404:         IF(!intern->u.file.stream) {
2405:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2406:             return;
2407:         }
2408:
2409:         IF (spl_filesystem_file_read(intern, 0) == FAILURE) {
2410:             RETURN_FALSE;
2411:         }
2412:
2413:         RETURN_STRING(intern->u.file.current_line, intern->u.file.current_line_len);
2414:     } /* }}} */
2415:
2416:     /* {{{ proto string SplFileObject::current()
2417:     Return current line from file */
2418:     SPL_METHOD(SplFileObject, current)
2419:     {
2420:         spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2421:
2422:         IF (zend_parse_parameters_none() == FAILURE) {
2423:             return;
2424:         }
2425:
2426:         IF(!intern->u.file.stream) {
2427:             zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2428:             return;
2429:         }
2430:
2431:         IF (intern->u.file.current_line && !IS_HUNDERF(intern->u.file.current_val)) {
2432:             spl_filesystem_file_read_line(intern, 1);
2433:         }
2434:
2435:         IF (intern->u.file.current_line && !IS_HUNDERF(intern->u.file.current_val)) {
2436:             RETURN_STRING(intern->u.file.current_line, intern->u.file.current_line_len);
2437:         } ELSE IF (!IS_HUNDERF(intern->u.file.current_val)) {
2438:             eval *value = &intern->u.file.current_val;
2439:             RETURN_ZVAL(value);
2440:         } ELSE {
2441:             RETURN_ZVAL(return_value, value);
2442:         }
2443:     }
2444:
2445:     /* {{{ proto void SplFileObject::setFlags(int flags)
2446:     Set file handling flags */
2447:     SPL_METHOD(SplFileObject, setFlags)
2448:     {
2449:         spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2450:
2451:         IF (zend_parse_parameters_none() == FAILURE) {
2452:             return;
2453:         }
2454:
2455:         IF (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &intern->flags) == FAILURE) {
2456:             return;
2457:         }
2458:
2459:         /* {{{ proto void SplFileObject::getFlags()
2460:         Get file handling flags */
2461:         SPL_METHOD(SplFileObject, getFlags)
2462:         {
2463:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2464:
2465:             IF (zend_parse_parameters_none() == FAILURE) {
2466:                 return;
2467:             }
2468:
2469:             RETURN_LONG(intern->flags & SPL_FILE_OBJECT_READ Ahead);
2470:         } /* }}} */
2471:
2472:         /* {{{ proto void SplFileObject::setMaxLineLen(int max_line)
2473:         Set maximum line length */
2474:         SPL_METHOD(SplFileObject, setMaxLineLen)
2475:         {
2476:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2477:
2478:             IF (zend_parse_parameters_none() == FAILURE) {
2479:                 return;
2480:             }
2481:
2482:             RETURN_LONG(intern->u.file.max_line_len);
2483:         } /* }}} */
2484:
2485:         /* {{{ proto void SplFileObject::setMaxLineLen(int max_line)
2486:         Set maximum line length */
2487:         SPL_METHOD(SplFileObject, getMaxLineLen)
2488:         {
2489:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2490:
2491:             IF (zend_parse_parameters_none() == FAILURE) {
2492:                 return;
2493:             }
2494:
2495:             IF (max_line < 0) {
2496:                 zend_throw_exception_ex(spl_ce_DomainException, 0, "Maximum line length must be greater than or equal zero");
2497:                 return;
2498:             }
2499:
2500:             intern->u.file.max_line_len = max_line;
2501:         } /* }}} */
2502:
2503:         /* {{{ proto void SplFileObject::getMaxLineLen()
2504:         Get maximum line length */
2505:         SPL_METHOD(SplFileObject, getMaxLineLen)
2506:         {
2507:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2508:
2509:             IF (zend_parse_parameters_none() == FAILURE) {
2510:                 return;
2511:             }
2512:
2513:             RETURN_LONG((zend_long)intern->u.file.max_line_len);
2514:         } /* }}} */
2515:
2516:         /* {{{ proto bool SplFileObject::hasChildren()
2517:         Return false */
2518:         SPL_METHOD(SplFileObject, hasChildren)
2519:         {
2520:             IF (zend_parse_parameters_none() == FAILURE) {
2521:                 return;
2522:             }
2523:
2524:             RETURN_FALSE;
2525:         } /* }}} */
2526:
2527:         /* {{{ proto bool SplFileObject::getChildren()
2528:         Read NULL */
2529:         SPL_METHOD(SplFileObject, getChildren)
2530:         {
2531:             IF (zend_parse_parameters_none() == FAILURE) {
2532:                 return;
2533:             }
2534:
2535:             /* return NULL */
2536:             /* }}} */
2537:
2538:         /* {{{ FileFunction */
2539:         DEFINE_FILEFUNCTION(func_name) \
2540:         {
2541:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis()); \
2542:             FileFunctionCall(func_name, ZEND_NUM_ARGS(), NULL); \
2543:         } \
2544:         /* }}} */
2545:
2546:         /* {{{ proto array SplFileObject::fgencv(string delimiter [, string enclosure [, escape = '\\']])
2547:         Return current line as csv */
2548:         SPL_METHOD(SplFileObject, fgencv)
2549:         {
2550:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2551:
2552:             char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
2553:             char *dlen = NULL, *enclo = NULL, *esc = NULL;
2554:             size_t d_len = 0, e_len = 0, esc_len = 0;
2555:
2556:             IF (zend_parse_parameters(ZEND_NUM_ARGS(), "ssss", &delim, &dlen, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2557:                 IF(!intern->u.file.stream) {
2558:                     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2559:                     return;
2560:                 }
2561:
2562:                 switch(ZEND_NUM_ARGS()) {
2563:                     case 0:
2564:                         IF (esc_len != 1) {
2565:                             php_error_docref(NULL, E_WARNING, "escape must be a character");
2566:                             RETURN_FALSE;
2567:                         }
2568:                         escape = esc[0];
2569:                         /* no break */
2570:                     case 1:
2571:                         IF (e_len != 1) {
2572:                             php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2573:                             RETURN_FALSE;
2574:                         }
2575:                         enclosure = enclo[0];
2576:                         /* no break */
2577:                     case 2:
2578:                         IF (d_len != 1) {
2579:                             php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2580:                             RETURN_FALSE;
2581:                         }
2582:                         delimiter = delim[0];
2583:                         /* no break */
2584:                     case 3:
2585:                         break;
2586:                 }
2587:
2588:                 spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
2589:             }
2590:
2591:             /* }}} */
2592:
2593:         /* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]])
2594:         Output a field array as a CSV line */
2595:         SPL_METHOD(SplFileObject, fputcsv)
2596:         {
2597:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2598:
2599:             char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
2600:             char *dlen = NULL, *enclo = NULL, *esc = NULL;
2601:             size_t d_len = 0, e_len = 0, esc_len = 0;
2602:
2603:             zend_long val;
2604:             eval *fields = NULL;
2605:
2606:             IF (zend_parse_parameters(ZEND_NUM_ARGS(), "assss", &fields, &dlen, &delim, &dlen, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2607:                 switch(ZEND_NUM_ARGS()) {
2608:                     case 0:
2609:                         IF (esc_len != 1) {
2610:                             php_error_docref(NULL, E_WARNING, "escape must be a character");
2611:                             RETURN_FALSE;
2612:                         }
2613:                         escape = esc[0];
2614:                         /* no break */
2615:                     case 1:
2616:                         IF (e_len != 1) {
2617:                             php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2618:                             RETURN_FALSE;
2619:                         }
2620:                         enclosure = enclo[0];
2621:                         /* no break */
2622:                     case 2:
2623:                         IF (d_len != 1) {
2624:                             php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2625:                             RETURN_FALSE;
2626:                         }
2627:                         delimiter = delim[0];
2628:                         /* no break */
2629:                     case 3:
2630:                         break;
2631:                 }
2632:
2633:                 spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
2634:             }
2635:
2636:             /* }}} */
2637:
2638:         /* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]])
2639:         Output a field array as a CSV line */
2640:         SPL_METHOD(SplFileObject, fputcsv)
2641:         {
2642:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2643:
2644:             char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
2645:             char *dlen = NULL, *enclo = NULL, *esc = NULL;
2646:             size_t d_len = 0, e_len = 0, esc_len = 0;
2647:
2648:             zend_long val;
2649:             eval *fields = NULL;
2650:
2651:             IF (zend_parse_parameters(ZEND_NUM_ARGS(), "assss", &fields, &dlen, &delim, &dlen, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2652:                 switch(ZEND_NUM_ARGS()) {
2653:                     case 0:
2654:                         IF (esc_len != 1) {
2655:                             php_error_docref(NULL, E_WARNING, "escape must be a character");
2656:                             RETURN_FALSE;
2657:                         }
2658:                         escape = esc[0];
2659:                         /* no break */
2660:                     case 1:
2661:                         IF (e_len != 1) {
2662:                             php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2663:                             RETURN_FALSE;
2664:                         }
2665:                         enclosure = enclo[0];
2666:                         /* no break */
2667:                     case 2:
2668:                         IF (d_len != 1) {
2669:                             php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2670:                             RETURN_FALSE;
2671:                         }
2672:                         delimiter = delim[0];
2673:                         /* no break */
2674:                     case 3:
2675:                         break;
2676:                 }
2677:
2678:                 spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
2679:             }
2680:
2681:             /* }}} */
2682:
2683:         /* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]])
2684:         Output a field array as a CSV line */
2685:         SPL_METHOD(SplFileObject, fputcsv)
2686:         {
2687:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2688:
2689:             char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
2690:             char *dlen = NULL, *enclo = NULL, *esc = NULL;
2691:             size_t d_len = 0, e_len = 0, esc_len = 0;
2692:
2693:             zend_long val;
2694:             eval *fields = NULL;
2695:
2696:             IF (zend_parse_parameters(ZEND_NUM_ARGS(), "assss", &fields, &dlen, &delim, &dlen, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2697:                 switch(ZEND_NUM_ARGS()) {
2698:                     case 0:
2699:                         IF (esc_len != 1) {
2700:                             php_error_docref(NULL, E_WARNING, "escape must be a character");
2701:                             RETURN_FALSE;
2702:                         }
2703:                         escape = esc[0];
2704:                         /* no break */
2705:                     case 1:
2706:                         IF (e_len != 1) {
2707:                             php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2708:                             RETURN_FALSE;
2709:                         }
2710:                         enclosure = enclo[0];
2711:                         /* no break */
2712:                     case 2:
2713:                         IF (d_len != 1) {
2714:                             php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2715:                             RETURN_FALSE;
2716:                         }
2717:                         delimiter = delim[0];
2718:                         /* no break */
2719:                     case 3:
2720:                         break;
2721:                 }
2722:
2723:                 spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
2724:             }
2725:
2726:             /* }}} */
2727:
2728:         /* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]])
2729:         Output a field array as a CSV line */
2730:         SPL_METHOD(SplFileObject, fputcsv)
2731:         {
2732:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2733:
2734:             char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
2735:             char *dlen = NULL, *enclo = NULL, *esc = NULL;
2736:             size_t d_len = 0, e_len = 0, esc_len = 0;
2737:
2738:             zend_long val;
2739:             eval *fields = NULL;
2740:
2741:             IF (zend_parse_parameters(ZEND_NUM_ARGS(), "assss", &fields, &dlen, &delim, &dlen, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2742:                 switch(ZEND_NUM_ARGS()) {
2743:                     case 0:
2744:                         IF (esc_len != 1) {
2745:                             php_error_docref(NULL, E_WARNING, "escape must be a character");
2746:                             RETURN_FALSE;
2747:                         }
2748:                         escape = esc[0];
2749:                         /* no break */
2750:                     case 1:
2751:                         IF (e_len != 1) {
2752:                             php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2753:                             RETURN_FALSE;
2754:                         }
2755:                         enclosure = enclo[0];
2756:                         /* no break */
2757:                     case 2:
2758:                         IF (d_len != 1) {
2759:                             php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2760:                             RETURN_FALSE;
2761:                         }
2762:                         delimiter = delim[0];
2763:                         /* no break */
2764:                     case 3:
2765:                         break;
2766:                 }
2767:
2768:                 spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
2769:             }
2770:
2771:             /* }}} */
2772:
2773:         /* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]])
2774:         Output a field array as a CSV line */
2775:         SPL_METHOD(SplFileObject, fputcsv)
2776:         {
2777:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2778:
2779:             char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
2780:             char *dlen = NULL, *enclo = NULL, *esc = NULL;
2781:             size_t d_len = 0, e_len = 0, esc_len = 0;
2782:
2783:             zend_long val;
2784:             eval *fields = NULL;
2785:
2786:             IF (zend_parse_parameters(ZEND_NUM_ARGS(), "assss", &fields, &dlen, &delim, &dlen, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2787:                 switch(ZEND_NUM_ARGS()) {
2788:                     case 0:
2789:                         IF (esc_len != 1) {
2790:                             php_error_docref(NULL, E_WARNING, "escape must be a character");
2791:                             RETURN_FALSE;
2792:                         }
2793:                         escape = esc[0];
2794:                         /* no break */
2795:                     case 1:
2796:                         IF (e_len != 1) {
2797:                             php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2798:                             RETURN_FALSE;
2799:                         }
2800:                         enclosure = enclo[0];
2801:                         /* no break */
2802:                     case 2:
2803:                         IF (d_len != 1) {
2804:                             php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2805:                             RETURN_FALSE;
2806:                         }
2807:                         delimiter = delim[0];
2808:                         /* no break */
2809:                     case 3:
2810:                         break;
2811:                 }
2812:
2813:                 spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
2814:             }
2815:
2816:             /* }}} */
2817:
2818:         /* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]])
2819:         Output a field array as a CSV line */
2820:         SPL_METHOD(SplFileObject, fputcsv)
2821:         {
2822:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2823:
2824:             char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
2825:             char *dlen = NULL, *enclo = NULL, *esc = NULL;
2826:             size_t d_len = 0, e_len = 0, esc_len = 0;
2827:
2828:             zend_long val;
2829:             eval *fields = NULL;
2830:
2831:             IF (zend_parse_parameters(ZEND_NUM_ARGS(), "assss", &fields, &dlen, &delim, &dlen, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2832:                 switch(ZEND_NUM_ARGS()) {
2833:                     case 0:
2834:                         IF (esc_len != 1) {
2835:                             php_error_docref(NULL, E_WARNING, "escape must be a character");
2836:                             RETURN_FALSE;
2837:                         }
2838:                         escape = esc[0];
2839:                         /* no break */
2840:                     case 1:
2841:                         IF (e_len != 1) {
2842:                             php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2843:                             RETURN_FALSE;
2844:                         }
2845:                         enclosure = enclo[0];
2846:                         /* no break */
2847:                     case 2:
2848:                         IF (d_len != 1) {
2849:                             php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2850:                             RETURN_FALSE;
2851:                         }
2852:                         delimiter = delim[0];
2853:                         /* no break */
2854:                     case 3:
2855:                         break;
2856:                 }
2857:
2858:                 spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
2859:             }
2860:
2861:             /* }}} */
2862:
2863:         /* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]])
2864:         Output a field array as a CSV line */
2865:         SPL_METHOD(SplFileObject, fputcsv)
2866:         {
2867:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2868:
2869:             char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
2870:             char *dlen = NULL, *enclo = NULL, *esc = NULL;
2871:             size_t d_len = 0, e_len = 0, esc_len = 0;
2872:
2873:             zend_long val;
2874:             eval *fields = NULL;
2875:
2876:             IF (zend_parse_parameters(ZEND_NUM_ARGS(), "assss", &fields, &dlen, &delim, &dlen, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2877:                 switch(ZEND_NUM_ARGS()) {
2878:                     case 0:
2879:                         IF (esc_len != 1) {
2880:                             php_error_docref(NULL, E_WARNING, "escape must be a character");
2881:                             RETURN_FALSE;
2882:                         }
2883:                         escape = esc[0];
2884:                         /* no break */
2885:                     case 1:
2886:                         IF (e_len != 1) {
2887:                             php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2888:                             RETURN_FALSE;
2889:                         }
2890:                         enclosure = enclo[0];
2891:                         /* no break */
2892:                     case 2:
2893:                         IF (d_len != 1) {
2894:                             php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2895:                             RETURN_FALSE;
2896:                         }
2897:                         delimiter = delim[0];
2898:                         /* no break */
2899:                     case 3:
2900:                         break;
2901:                 }
2902:
2903:                 spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
2904:             }
2905:
2906:             /* }}} */
2907:
2908:         /* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]])
2909:         Output a field array as a CSV line */
2910:         SPL_METHOD(SplFileObject, fputcsv)
2911:         {
2912:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2913:
2914:             char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
2915:             char *dlen = NULL, *enclo = NULL, *esc = NULL;
2916:             size_t d_len = 0, e_len = 0, esc_len = 0;
2917:
2918:             zend_long val;
2919:             eval *fields = NULL;
2920:
2921:             IF (zend_parse_parameters(ZEND_NUM_ARGS(), "assss", &fields, &dlen, &delim, &dlen, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2922:                 switch(ZEND_NUM_ARGS()) {
2923:                     case 0:
2924:                         IF (esc_len != 1) {
2925:                             php_error_docref(NULL, E_WARNING, "escape must be a character");
2926:                             RETURN_FALSE;
2927:                         }
2928:                         escape = esc[0];
2929:                         /* no break */
2930:                     case 1:
2931:                         IF (e_len != 1) {
2932:                             php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2933:                             RETURN_FALSE;
2934:                         }
2935:                         enclosure = enclo[0];
2936:                         /* no break */
2937:                     case 2:
2938:                         IF (d_len != 1) {
2939:                             php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2940:                             RETURN_FALSE;
2941:                         }
2942:                         delimiter = delim[0];
2943:                         /* no break */
2944:                     case 3:
2945:                         break;
2946:                 }
2947:
2948:                 spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
2949:             }
2950:
2951:             /* }}} */
2952:
2953:         /* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]])
2954:         Output a field array as a CSV line */
2955:         SPL_METHOD(SplFileObject, fputcsv)
2956:         {
2957:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
2958:
2959:             char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
2960:             char *dlen = NULL, *enclo = NULL, *esc = NULL;
2961:             size_t d_len = 0, e_len = 0, esc_len = 0;
2962:
2963:             zend_long val;
2964:             eval *fields = NULL;
2965:
2966:             IF (zend_parse_parameters(ZEND_NUM_ARGS(), "assss", &fields, &dlen, &delim, &dlen, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2967:                 switch(ZEND_NUM_ARGS()) {
2968:                     case 0:
2969:                         IF (esc_len != 1) {
2970:                             php_error_docref(NULL, E_WARNING, "escape must be a character");
2971:                             RETURN_FALSE;
2972:                         }
2973:                         escape = esc[0];
2974:                         /* no break */
2975:                     case 1:
2976:                         IF (e_len != 1) {
2977:                             php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2978:                             RETURN_FALSE;
2979:                         }
2980:                         enclosure = enclo[0];
2981:                         /* no break */
2982:                     case 2:
2983:                         IF (d_len != 1) {
2984:                             php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2985:                             RETURN_FALSE;
2986:                         }
2987:                         delimiter = delim[0];
2988:                         /* no break */
2989:                     case 3:
2990:                         break;
2991:                 }
2992:
2993:                 spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
2994:             }
2995:
2996:             /* }}} */
2997:
2998:         /* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]])
2999:         Output a field array as a CSV line */
3000:         SPL_METHOD(SplFileObject, fputcsv)
3001:         {
3002:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
3003:
3004:             char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
3005:             char *dlen = NULL, *enclo = NULL, *esc = NULL;
3006:             size_t d_len = 0, e_len = 0, esc_len = 0;
3007:
3008:             zend_long val;
3009:             eval *fields = NULL;
3010:
3011:             IF (zend_parse_parameters(ZEND_NUM_ARGS(), "assss", &fields, &dlen, &delim, &dlen, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
3012:                 switch(ZEND_NUM_ARGS()) {
3013:                     case 0:
3014:                         IF (esc_len != 1) {
3015:                             php_error_docref(NULL, E_WARNING, "escape must be a character");
3016:                             RETURN_FALSE;
3017:                         }
3018:                         escape = esc[0];
3019:                         /* no break */
3020:                     case 1:
3021:                         IF (e_len != 1) {
3022:                             php_error_docref(NULL, E_WARNING, "enclosure must be a character");
3023:                             RETURN_FALSE;
3024:                         }
3025:                         enclosure = enclo[0];
3026:                         /* no break */
3027:                     case 2:
3028:                         IF (d_len != 1) {
3029:                             php_error_docref(NULL, E_WARNING, "delimiter must be a character");
3030:                             RETURN_FALSE;
3031:                         }
3032:                         delimiter = delim[0];
3033:                         /* no break */
3034:                     case 3:
3035:                         break;
3036:                 }
3037:
3038:                 spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
3039:             }
3040:
3041:             /* }}} */
3042:
3043:         /* {{{ proto int SplFileObject::fputcsv(array fields, [string delimiter [, string enclosure [, string escape]])
3044:         Output a field array as a CSV line */
3045:         SPL_METHOD(SplFileObject, fputcsv)
3046:         {
3047:             spl_filesystem_object *intern = _Z_SPFILESYSTEM_P(getThis());
3048:
3049:             char delimiter = intern->u.file.delimiter, enclosure = intern->u.file.enclosure, escape = intern->u.file.escape;
3050:             char *dlen = NULL, *enclo = NULL, *esc = NULL;
3051:             size_t d_len = 0, e_len = 0, esc_len = 0;
3052:
3053:             zend_long val;
3054:             eval *fields = NULL;
3055:
3056:             IF (zend_parse_parameters(ZEND_NUM_ARGS(), "assss", &fields, &dlen, &delim, &dlen, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
3057:                 switch(ZEND_NUM_ARGS()) {
3058:                     case 0:
3059:                         IF (esc_len != 1) {
3060:                             php_error_docref(NULL, E_WARNING, "escape must be a character");
3061:                             RETURN_FALSE;
3062:                         }
3063:                         escape = esc[0];
3064:                         /* no break */
3065:                     case 1:
3066:                         IF (e_len != 1) {
3067:                             php_error_docref(NULL, E_WARNING, "enclosure must be a character");
3068:                             RETURN_FALSE;
3069:                         }
3070:                         enclosure = enclo[0];
3071:                         /* no break */
3072:                     case 2:
3073:                         IF (d_len != 1) {
3074:                             php_error_docref(NULL, E_WARNING, "delimiter must be a character");
3075:                             RETURN_FALSE;
3076:                         }
3077:                         delimiter = delim[0];
3078:                         /* no break */
3079:                     case 3:
3080:                         break;
3081:                 }
3082:
3083:                 spl_filesystem_file_read_csv(intern, delimiter, enclosure, escape, return_value);
3084:             }
3085:
3086:             /* }}} */
3087:
3088:         /* {{{ proto int SplFileObject::fputcsv(array fields,
```

```
2629:     php_error_docref(NULL, E_WARNING, "escape must be a character");
2630:     RETURN_FALSE;
2631: }
2632: escape = esc[0];
2633: /* no break */
2634: case 3:
2635:     if (d_len != 1) {
2636:         php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2637:         RETURN_FALSE;
2638:     }
2639:     enclosure = enclo[0];
2640:     /* no break */
2641: case 2:
2642:     if (d_len != 1) {
2643:         php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2644:         RETURN_FALSE;
2645:     }
2646:     delimiter = delim[0];
2647:     /* no break */
2648: case 1:
2649: case 0:
2650:     /*break;
2651:     */
2652:     ret = php_fputcsv(intern->u.file.stream, fields, delimiter, enclosure, escape);
2653:     RETURN_LONG(ret);
2654: }
2655: /* }}} */
2656:
2657:
2658: /* {{{ proto void SplFileObject::setCsvControl([string delimiter [, string enclosure [, string escape]])
2659:    Set the delimiter, enclosure and escape character used in fgetcsv */
2660: SPL_METHOD(SplFileObject, setCsvControl)
2661: {
2662:     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2663:     char delimiter = ',';
2664:     char *enclo = NULL, *enclo = NULL, *esc = NULL;
2665:     size_t d_len = 0, e_len = 0, esc_len = 0;
2666:
2667:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "[sss]", &delim, &d_len, &enclo, &e_len, &esc, &esc_len) == SUCCESS) {
2668:         switch(ZEND_NUM_ARGS()) {
2669:             case 3:
2670:                 if (d_len != 1) {
2671:                     php_error_docref(NULL, E_WARNING, "escape must be a character");
2672:                     RETURN_FALSE;
2673:                 }
2674:                 escape = esc[0];
2675:                 /* no break */
2676:             case 2:
2677:                 if (e_len != 1) {
2678:                     php_error_docref(NULL, E_WARNING, "enclosure must be a character");
2679:                     RETURN_FALSE;
2680:                 }
2681:                 enclosure = enclo[0];
2682:                 /* no break */
2683:             case 1:
2684:                 if (d_len != 1) {
2685:                     php_error_docref(NULL, E_WARNING, "delimiter must be a character");
2686:                     RETURN_FALSE;
2687:                 }
2688:                 delimiter = delim[0];
2689:                 /* no break */
2690:             case 0:
2691:                 /*break;
2692:                 */
2693:             }
2694:             intern->u.file.delimiter = delimiter;
2695:             intern->u.file.enclosure = enclosure;
2696:             intern->u.file.escape = escape;
2697:         }
2698:         /* }}} */
2699:
2700:     /* {{{ proto array SplFileObject::getCsvControl()
2701:        Get the delimiter, enclosure and escape character used in fgetcsv */
2702:     SPL_METHOD(SplFileObject, getCsvControl)
2703:     {
2704:         spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2705:         char delimiter[2], enclosure[2], escape[2];
2706:         array_init(return_value);
2707:
2708:         delimiter[0] = intern->u.file.delimiter;
2709:         delimiter[1] = '\0';
2710:         enclosure[0] = intern->u.file.enclosure;
2711:         enclosure[1] = '\0';
2712:         escape[0] = intern->u.file.escape;
2713:         escape[1] = '\0';
2714:
2715:         add_next_index_string(return_value, delimiter);
2716:         add_next_index_string(return_value, enclosure);
2717:         add_next_index_string(return_value, escape);
2718:     }
2719:     /* }}} */
2720:
2721:     /* {{{ proto bool SplFileObject::flock(int operation [, int wouldblock])
2722:        Portable file locking */
2723:     SPL_METHOD(SplFileObject, flock)
2724:     {
2725:         /* {{{ proto bool SplFileObject::ftruncate()
2726:            Flush the file */
2727:         SPL_METHOD(SplFileObject, ftruncate)
2728:         {
2729:             spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2730:
2731:             if (intern->u.file.stream) {
2732:                 zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2733:                 return;
2734:             }
2735:
2736:             RETURN_BOOL(php_stream_flush(intern->u.file.stream));
2737:         }
2738:         /* }}} */
2739:
2740:         /* {{{ proto int SplFileObject::ftell()
2741:            Return current file position */
2742:         SPL_METHOD(SplFileObject, ftell)
2743:         {
2744:             spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2745:             zend_long ret;
2746:
2747:             if (intern->u.file.stream) {
2748:                 zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2749:                 return;
2750:             }
2751:
2752:             ret = php_stream_tell(intern->u.file.stream);
2753:
2754:             if (ret == -1) {
2755:                 RETURN_FALSE;
2756:             } else {
2757:                 RETURN_LONG(ret);
2758:             }
2759:         }
2760:         /* }}} */
2761:
2762:         /* {{{ proto int SplFileObject::fseek(int pos [, int whence = SEEK_SET])
2763:            Return current file position */
2764:         SPL_METHOD(SplFileObject, fseek)
2765:         {
2766:             spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2767:             zend_long pos, whence = SEEK_SET;
2768:
2769:             if (zend_parse_parameters(ZEND_NUM_ARGS(), "[iI]", &pos, &whence) == FAILURE) {
2770:                 return;
2771:             }
2772:
2773:             if (intern->u.file.stream) {
2774:                 zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2775:                 return;
2776:             }
2777:
2778:             spl_filesystem_file_free_line(intern);
2779:             RETURN_LONG(php_stream_seek(intern->u.file.stream, pos, (int)whence));
2780:         }
2781:         /* }}} */
2782:
2783:         /* {{{ proto int SplFileObject::fgetc()
2784:            Get a character from the file */
2785:         SPL_METHOD(SplFileObject, fgetc)
2786:         {
2787:             spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2788:             char buf[2];
2789:             int result;
2790:
2791:             if (intern->u.file.stream) {
2792:                 zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2793:                 return;
2794:             }
2795:
2796:             spl_filesystem_file_free_line(intern);
2797:             result = php_stream_getc(intern->u.file.stream);
2798:
2799:             if (result == EOF) {
2800:                 RETURN_FALSE;
2801:             } else {
2802:                 if (result == '\n') {
2803:                     intern->u.file.current_line_num++;
2804:                 }
2805:                 buf[0] = result;
2806:                 buf[1] = '\0';
2807:                 RETURN_STRING(buf, 1);
2808:             }
2809:         }
2810:         /* }}} */
2811:
2812:         /* {{{ proto string SplFileObject::fgets([string allowable_tags])
2813:            Get a line from file pointer and strip HTML tags */
2814:         SPL_METHOD(SplFileObject, fgets)
2815:         {
2816:             spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2817:             zend_long arg2;
2818:
2819:             if (intern->u.file.stream) {
2820:                 zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2821:                 return;
2822:             }
2823:
2824:             if (intern->u.file.max_line_len > 0) {
2825:                 ZVAL_LONG(&arg2, intern->u.file.max_line_len);
2826:             } else {
2827:                 ZVAL_LONG(&arg2, 1024);
2828:             }
2829:
2830:             FileFunctionCall(fgets, ZEND_NUM_ARGS(), &arg2);
2831:             /* }}} */
2832:
2833:             /* {{{ proto int SplFileObject::fpassthru()
2834:                Output all remaining data from a file pointer */
2835:             SPL_METHOD(SplFileObject, fpassthru)
2836:             {
2837:                 spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2838:
2839:                 if (intern->u.file.stream) {
2840:                     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2841:                     return;
2842:                 }
2843:
2844:                 RETURN_LONG(php_stream_passthru(intern->u.file.stream));
2845:             }
2846:             /* }}} */
2847:
2848:             /* {{{ proto bool SplFileObject::fscanf(string format [, string ...])
2849:                Implements a mostly ANSI compatible fscanf() */
2850:             SPL_METHOD(SplFileObject, fscanf)
2851:             {
2852:                 spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2853:
2854:                 if (intern->u.file.stream) {
2855:                     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2856:                     return;
2857:                 }
2858:
2859:                 spl_filesystem_file_free_line(intern);
2860:                 intern->u.file.current_line_num++;
2861:
2862:                 FileFunctionCall(fscanf, ZEND_NUM_ARGS(), NULL);
2863:             }
2864:             /* }}} */
2865:
2866:             /* {{{ proto mixed SplFileObject::fwrite(string str [, int length])
2867:                Binary-safe file write */
2868:             SPL_METHOD(SplFileObject, fwrite)
2869:             {
2870:                 spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2871:                 char *str;
2872:                 size_t str_len;
2873:                 zend_long length = 0;
2874:
2875:                 if (zend_parse_parameters(ZEND_NUM_ARGS(), ["s", "i", "i", "i"], &str, &str_len, &length) == FAILURE) {
2876:                     return;
2877:                 }
2878:
2879:                 if (intern->u.file.stream) {
2880:                     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2881:                     return;
2882:                 }
2883:
2884:                 if (ZEND_NUM_ARGS() > 1) {
2885:                     if (length >= 0) {
2886:                         str_len = MIN((size_t)length, str_len);
2887:                     } else {
2888:                         /* Negative length given, nothing to write */
2889:                         str_len = 0;
2890:                     }
2891:                 }
2892:
2893:                 if (!str) {
2894:                     RETURN_LONG(0);
2895:                 }
2896:
2897:                 RETURN_LONG(php_stream_write(intern->u.file.stream, str, str_len));
2898:             }
2899:             /* }}} */
2900:
2901:             SPL_METHOD(SplFileObject, fread)
2902:             {
2903:                 spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2904:                 zend_long length = 0;
2905:
2906:                 if (zend_parse_parameters(ZEND_NUM_ARGS(), ["i", "i"], &length) == FAILURE) {
2907:                     return;
2908:                 }
2909:
2910:                 if (intern->u.file.stream) {
2911:                     zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2912:                     return;
2913:                 }
2914:
2915:                 if (length <= 0) {
2916:                     php_error_docref(NULL, E_WARNING, "Length parameter must be greater than 0");
2917:                     RETURN_FALSE;
2918:                 }
2919:
2920:                 ZVAL_NEW_STR(return_value, zend_string_alloc(length, 0));
2921:                 Z_STRLEN_P(return_value) = php_stream_read(intern->u.file.stream, Z_STRVAL_P(return_value), length);
2922:
2923:                 /* needed because read/read/gread doesn't put a null at the end */
2924:                 Z_STRVAL_P(return_value)[Z_STRLEN_P(return_value)] = 0;
2925:             }
2926:
2927:             /* {{{ proto bool SplFileObject::fstat()
2928:                Stat() on a filehandle */
2929:             FileFunction(fstat)
2930:             {
2931:                 /* }}} */
2932:
2933:                 /* {{{ proto bool SplFileObject::ftruncate(int size)
2934:                    Truncate file to 'size' length */
2935:                 SPL_METHOD(SplFileObject, ftruncate)
2936:                 {
2937:                     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2938:                     zend_long size;
2939:
2940:                     if (zend_parse_parameters(ZEND_NUM_ARGS(), ["i"], &size) == FAILURE) {
2941:                         return;
2942:                     }
2943:
2944:                     if (intern->u.file.stream) {
2945:                         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2946:                         return;
2947:                     }
2948:
2949:                     if (php_stream_truncate_supported(intern->u.file.stream) {
2950:                         zend_throw_exception_ex(spl_ce_LogicException, 0, "Can't truncate file to", intern->u.file.name);
2951:                         RETURN_FALSE;
2952:                     }
2953:
2954:                     RETURN_BOOL(0 == php_stream_truncate_set_size(intern->u.file.stream, size));
2955:                 }
2956:                 /* }}} */
2957:
2958:                 /* {{{ proto void SplFileObject::seek(int line_pos)
2959:                    Seek to specified line */
2960:                 SPL_METHOD(SplFileObject, seek)
2961:                 {
2962:                     spl_filesystem_object *intern = _SPL_FILESYSTEM_P(getThis());
2963:                     zend_long line_pos;
2964:
2965:                     if (zend_parse_parameters(ZEND_NUM_ARGS(), ["i"], &line_pos) == FAILURE) {
2966:                         return;
2967:                     }
2968:
2969:                     if (intern->u.file.stream) {
2970:                         zend_throw_exception_ex(spl_ce_RuntimeException, 0, "Object not initialized");
2971:                         return;
2972:                     }
2973:
2974:                     if (line_pos < 0) {
2975:                         zend_throw_exception_ex(spl_ce_LogicException, 0, "Can't seek file to negative line", ZEND_LONG_PWT, intern->u.file.name, line_pos);
2976:                     }
2977:
2978:                     spl_filesystem_file_rewind(getThis(), intern);
2979:
2980:                     while(intern->u.file.current_line_num < line_pos) {
2981:                         if (spl_filesystem_file_read_line(getThis()) == FAILURE) {
2982:                             break;
2983:                         }
2984:                     }
2985:                 }
2986:                 /* }}} */
2987:
2988:                 /* {{{ Function/Class/Method definitions */
2989:                 ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_construct, 0, 1)
2990:                     ZEND_ARG_INFO(0, file_name)
2991:                     ZEND_ARG_INFO(0, open_mode)
2992:                     ZEND_ARG_INFO(0, use_include_path)
2993:                     ZEND_ARG_INFO(0, context)
2994:                 ZEND_END_ARG_INFO()
2995:
2996:                 ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_setFlags, 0)
2997:                     ZEND_ARG_INFO(0, flags)
2998:                 ZEND_END_ARG_INFO()
2999:
3000:                 ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_setMaxLineLen, 0)
3001:                     ZEND_ARG_INFO(0, max_len)
3002:                 ZEND_END_ARG_INFO()
3003:
3004:                 ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fgetcsv, 0, 0)
3005:                     ZEND_ARG_INFO(0, delimiter)
3006:                     ZEND_ARG_INFO(0, enclosure)
3007:                     ZEND_ARG_INFO(0, escape)
3008:                 ZEND_END_ARG_INFO()
3009:             }
3010:         }
3011:     }
3012: }
```



```
3005: ZEND_ARG_INFO(0, escape)
3006: ZEND_END_ARG_INFO()
3007:
3008: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fputcsv, 0, 0, 1)
3009: ZEND_ARG_INFO(0, fields)
3010: ZEND_ARG_INFO(0, delimiter)
3011: ZEND_ARG_INFO(0, enclosure)
3012: ZEND_ARG_INFO(0, escape)
3013: ZEND_END_ARG_INFO()
3014:
3015: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_flock, 0, 0, 1)
3016: ZEND_ARG_INFO(0, operation)
3017: ZEND_ARG_INFO(1, wouldblock)
3018: ZEND_END_ARG_INFO()
3019:
3020: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fseek, 0, 0, 1)
3021: ZEND_ARG_INFO(0, pos)
3022: ZEND_ARG_INFO(0, whence)
3023: ZEND_END_ARG_INFO()
3024:
3025: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fgetas, 0, 0, 0)
3026: ZEND_ARG_INFO(0, allowable_tags)
3027: ZEND_END_ARG_INFO()
3028:
3029: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fscanf, 0, 0, 1)
3030: ZEND_ARG_INFO(0, format)
3031: ZEND_ARG_VARIADIC_INFO(1, vars)
3032: ZEND_END_ARG_INFO()
3033:
3034: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fwrite, 0, 0, 1)
3035: ZEND_ARG_INFO(0, str)
3036: ZEND_ARG_INFO(0, length)
3037: ZEND_END_ARG_INFO()
3038:
3039: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_fread, 0, 0, 1)
3040: ZEND_ARG_INFO(0, length)
3041: ZEND_END_ARG_INFO()
3042:
3043: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_ftruncate, 0, 0, 1)
3044: ZEND_ARG_INFO(0, size)
3045: ZEND_END_ARG_INFO()
3046:
3047: ZEND_BEGIN_ARG_INFO_EX(arginfo_file_object_seek, 0, 0, 1)
3048: ZEND_ARG_INFO(0, line_pos)
3049: ZEND_END_ARG_INFO()
3050:
3051: static const zend_function_entry spl_SplFileInfo_functions[] = {
3052:     SPL_ME(SplFileInfo, __construct, arginfo_file_object__construct, ZEND_ACC_PUBLIC)
3053:     SPL_ME(SplFileInfo, rewind, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3054:     SPL_ME(SplFileInfo, eof, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3055:     SPL_ME(SplFileInfo, valid, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3056:     SPL_ME(SplFileInfo, fgetc, arginfo_file_object_fgetc, ZEND_ACC_PUBLIC)
3057:     SPL_ME(SplFileInfo, fgetcsv, arginfo_file_object_fgetcsv, ZEND_ACC_PUBLIC)
3058:     SPL_ME(SplFileInfo, fputc, arginfo_file_object_fputc, ZEND_ACC_PUBLIC)
3059:     SPL_ME(SplFileInfo, setAccessControl, arginfo_file_object_fputcsv, ZEND_ACC_PUBLIC)
3060:     SPL_ME(SplFileInfo, getAccessControl, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3061:     SPL_ME(SplFileInfo, flock, arginfo_file_object_flock, ZEND_ACC_PUBLIC)
3062:     SPL_ME(SplFileInfo, fflush, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3063:     SPL_ME(SplFileInfo, ftruncate, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3064:     SPL_ME(SplFileInfo, fseek, arginfo_file_object_fseek, ZEND_ACC_PUBLIC)
3065:     SPL_ME(SplFileInfo, fgetc, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3066:     SPL_ME(SplFileInfo, fpassthru, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3067:     SPL_ME(SplFileInfo, fgetas, arginfo_file_object_fgetas, ZEND_ACC_PUBLIC)
3068:     SPL_ME(SplFileInfo, fscanf, arginfo_file_object_fscanf, ZEND_ACC_PUBLIC)
3069:     SPL_ME(SplFileInfo, fwrite, arginfo_file_object_fwrite, ZEND_ACC_PUBLIC)
3070:     SPL_ME(SplFileInfo, fread, arginfo_file_object_fread, ZEND_ACC_PUBLIC)
3071:     SPL_ME(SplFileInfo, fstat, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3072:     SPL_ME(SplFileInfo, truncate, arginfo_file_object_ftruncate, ZEND_ACC_PUBLIC)
3073:     SPL_ME(SplFileInfo, current, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3074:     SPL_ME(SplFileInfo, key, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3075:     SPL_ME(SplFileInfo, next, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3076:     SPL_ME(SplFileInfo, setFlags, arginfo_file_object_setFlags, ZEND_ACC_PUBLIC)
3077:     SPL_ME(SplFileInfo, getFlags, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3078:     SPL_ME(SplFileInfo, setMaxLineLen, arginfo_file_object_setMaxLineLen, ZEND_ACC_PUBLIC)
3079:     SPL_ME(SplFileInfo, getMaxLineLen, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3080:     SPL_ME(SplFileInfo, getChilden, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3081:     SPL_ME(SplFileInfo, getChilden, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3082:     SPL_ME(SplFileInfo, seek, arginfo_file_object_seek, ZEND_ACC_PUBLIC)
3083:     /* mappings */
3084:     SPL_ME(SplFileInfo, getCurrentLine, SplFileInfo, fgetc, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3085:     SPL_ME(SplFileInfo, __toString, SplFileInfo, current, arginfo_splfileinfo_void, ZEND_ACC_PUBLIC)
3086:     PHP_FE_END
3087: };
3088:
3089: ZEND_BEGIN_ARG_INFO_EX(arginfo_temp_file_object__construct, 0, 0, 0)
3090: ZEND_ARG_INFO(0, max_memory)
3091: ZEND_END_ARG_INFO()
3092:
3093: static const zend_function_entry spl_SplTempFileObject_functions[] = {
3094:     SPL_ME(SplTempFileObject, __construct, arginfo_temp_file_object__construct, ZEND_ACC_PUBLIC)
3095:     PHP_FE_END
3096: };
3097: /* }}} */
3098:
3099: /* {{{ PHP_MINIT_FUNCTION(spl_directory)
3100: */
3101: /* {{{ PHP_MINIT_FUNCTION(spl_directory)
3102: */
3103: REGISTER_SPL_STD_CLASS_EX(SplFileInfo, spl_filesystem_object_new, spl_SplFileInfo_functions);
3104: memory(spl_filesystem_object_handlers, zend_get_object_handlers(), sizeof(zend_object_handlers));
3105: spl_filesystem_object_handlers.offset = XOffsetOf(spl_filesystem_object, std);
3106: spl_filesystem_object_handlers.clone_obj = spl_filesystem_object_clone;
3107: spl_filesystem_object_handlers.cast_object = spl_filesystem_object_cast;
3108: spl_filesystem_object_handlers.get_debug_info = spl_filesystem_object_get_debug_info;
3109: spl_filesystem_object_handlers.dtor_obj = spl_filesystem_object_destroy_object;
3110: spl_filesystem_object_handlers.free_obj = spl_filesystem_object_free_storage;
3111: spl_ce_SplFileInfo->serialize = zend_class_serialize_deny;
3112: spl_ce_SplFileInfo->unserialize = zend_class_unserialize_deny;
3113:
3114:
3115: REGISTER_SPL_SUB_CLASS_EX(DirectoryIterator, SplFileInfo, spl_filesystem_object_new, spl_DirectoryIterator_functions);
3116: zend_class_implements(spl_ce_DirectoryIterator, 1, zend_ce_iterator);
3117: REGISTER_SPL_IMPLEMENTS(DirectoryIterator, SeekableIterator);
3118:
3119: spl_ce_DirectoryIterator->get_iterator = spl_filesystem_dir_get_iterator;
3120:
3121: REGISTER_SPL_SUB_CLASS_EX(FilesystemIterator, DirectoryIterator, spl_filesystem_object_new, spl_filesystem_iterator_functions);
3122:
3123: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_MODE_MASK", SPL_FILE_DIR_CURRENT_MODE_MASK);
3124: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_PATHNAME", SPL_FILE_DIR_CURRENT_AS_PATHNAME);
3125: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_FILEINFO", SPL_FILE_DIR_CURRENT_AS_FILEINFO);
3126: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "CURRENT_AS_SELF", SPL_FILE_DIR_CURRENT_AS_SELF);
3127: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_MODE_MASK", SPL_FILE_DIR_KEY_MODE_MASK);
3128: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_AS_PATHNAME", SPL_FILE_DIR_KEY_AS_PATHNAME);
3129: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "FOLLOW_SYMLINKS", SPL_FILE_DIR_FOLLOW_SYMLINKS);
3130: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "KEY_AS_FILENAME", SPL_FILE_DIR_KEY_AS_FILENAME);
3131: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "RECURSIVE_AND_ITER", SPL_FILE_DIR_KEY_AS_FILENAME(SPL_FILE_DIR_CURRENT_AS_FILEINFO);
3132: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "OTHER_MODE_MASK", SPL_FILE_DIR_OTHERS_MASK);
3133: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "SKIP_DOTS", SPL_FILE_DIR_SKIPDOTS);
3134: REGISTER_SPL_CLASS_CONST_LONG(FilesystemIterator, "UNIX_PATHS", SPL_FILE_DIR_UNIXPATHS);
3135:
3136: spl_ce_FilesystemIterator->get_iterator = spl_filesystem_tree_get_iterator;
3137:
3138: REGISTER_SPL_SUB_CLASS_EX(RecursiveDirectoryIterator, FilesystemIterator, spl_filesystem_object_new, spl_RecursiveDirectoryIterator_functions);
3139: REGISTER_SPL_IMPLEMENTS(RecursiveDirectoryIterator, RecursiveIterator);
3140:
3141: memory(spl_filesystem_object_check_handlers, spl_filesystem_object_handlers, sizeof(zend_object_handlers));
3142: spl_filesystem_object_check_handlers.get_method = spl_filesystem_object_get_method_check;
3143:
3144: #ifdef HAVE_GLOB
3145: REGISTER_SPL_SUB_CLASS_EX(GlobIterator, FilesystemIterator, spl_filesystem_object_new_check, spl_GlobIterator_functions);
3146: REGISTER_SPL_IMPLEMENTS(GlobIterator, Countable);
3147: #endif
3148:
3149: REGISTER_SPL_SUB_CLASS_EX(SplFileInfo, SplFileInfo, spl_filesystem_object_new_check, spl_SplFileInfo_functions);
3150: REGISTER_SPL_IMPLEMENTS(SplFileInfo, RecursiveIterator);
3151: REGISTER_SPL_IMPLEMENTS(SplFileInfo, SeekableIterator);
3152:
3153: REGISTER_SPL_CLASS_CONST_LONG(SplFileInfo, "DROP_NEW_LINE", SPL_FILE_OBJECT_DROP_NEW_LINE);
3154: REGISTER_SPL_CLASS_CONST_LONG(SplFileInfo, "READ_AHEAD", SPL_FILE_OBJECT_READ_AHEAD);
3155: REGISTER_SPL_CLASS_CONST_LONG(SplFileInfo, "READ_BUFFER", SPL_FILE_OBJECT_READ_BUFFER);
3156: REGISTER_SPL_CLASS_CONST_LONG(SplFileInfo, "READ_CSV", SPL_FILE_OBJECT_READ_CSV);
3157:
3158: REGISTER_SPL_SUB_CLASS_EX(SplTempFileObject, SplFileInfo, spl_filesystem_object_new_check, spl_SplTempFileObject_functions);
3159: #define SUCCESS;
3160: }
3161: /* }}} */
3162:
3163: /*
3164:  * Local variables:
3165:  * tab-width: 4
3166:  * c-basic-offset: 4
3167:  * End:
3168:  * vim600: noet sw=4 ts=4 fdm=marker
3169:  * vim600: noet sw=4 ts=4
3170:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | https://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_ITERATORS_H
22: #define SPL_ITERATORS_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26: #if HAVE_BUNDLED_PCRE
27: #include "ext/pcre/php_pcre.h"
28: #endif
29:
30: #define spl_ow_traversable send_ow_traversable
31: #define spl_ow_iterator send_ow_iterator
32: #define spl_ow_aggregate send_ow_aggregate
33: #define spl_ow_arrayaccess send_ow_arrayaccess
34: #define spl_ow_serializable send_ow_serializable
35: #define spl_ow_countable send_ow_countable
36:
37: #define PHPAPI zend_class_entry *spl_ow_recursiveiterator;
38: #define PHPAPI zend_class_entry *spl_ow_recursiveiteratoriterator;
39: #define PHPAPI zend_class_entry *spl_ow_recursiveiteratoriterator;
40: #define PHPAPI zend_class_entry *spl_ow_filteriterator;
41: #define PHPAPI zend_class_entry *spl_ow_recursivefilteriterator;
42: #define PHPAPI zend_class_entry *spl_ow_parentiterator;
43: #define PHPAPI zend_class_entry *spl_ow_weakableiterator;
44: #define PHPAPI zend_class_entry *spl_ow_limititerator;
45: #define PHPAPI zend_class_entry *spl_ow_cachingiterator;
46: #define PHPAPI zend_class_entry *spl_ow_recursivecachingiterator;
47: #define PHPAPI zend_class_entry *spl_ow_outertiterator;
48: #define PHPAPI zend_class_entry *spl_ow_iteratoriterator;
49: #define PHPAPI zend_class_entry *spl_ow_mohawinditerator;
50: #define PHPAPI zend_class_entry *spl_ow_infiniteiterator;
51: #define PHPAPI zend_class_entry *spl_ow_emptyiterator;
52: #define PHPAPI zend_class_entry *spl_ow_appenditerator;
53: #define PHPAPI zend_class_entry *spl_ow_requetiterator;
54: #define PHPAPI zend_class_entry *spl_ow_recursiverequetiterator;
55: #define PHPAPI zend_class_entry *spl_ow_callbackfilteriterator;
56: #define PHPAPI zend_class_entry *spl_ow_recursivecallbackfilteriterator;
57:
58: #define PHP_MINIT_FUNCTION(spl_iterators);
59:
60: #define PHP_FUNCTION(iterator_to_array);
61: #define PHP_FUNCTION(iterator_count);
62: #define PHP_FUNCTION(iterator_apply);
63:
64: typedef enum {
65:     DIT_Default = 0,
66:     DIT_Filteriterator = DIT_Default,
67:     DIT_RecursiveFilteriterator = DIT_Default,
68:     DIT_Parentiterator = DIT_Default,
69:     DIT_Limititerator,
70:     DIT_Cachingiterator,
71:     DIT_RecursiveCachingiterator,
72:     DIT_Iteratoriterator,
73:     DIT_Mohawinditerator,
74:     DIT_Infiniteiterator,
75:     DIT_Appenditerator,
76: #if HAVE_PCRE
77:     DIT_Requetiterator,
78:     DIT_RecursiveRequetiterator,
79: #endif
80:     DIT_CallbackFilteriterator,
81:     DIT_RecursiveCallbackFilteriterator,
82:     DIT_Unknown = 0
83: } dual_it_type;
84:
85: typedef enum {
86:     RIT_Default = 0,
87:     RIT_RecursiveIteratoriterator = RIT_Default,
88:     RIT_RecursiveTreeIterator,
89:     RIT_Unknown = 0
90: } recursive_it_type;
91:
92: enum {
93:     /* public */
94:     CIT_CALL_POSTING = 0x00000001,
95:     CIT_YOSTRING_USER_KEY = 0x00000002,
96:     CIT_YOSTRING_USER_CURRENT = 0x00000004,
97:     CIT_YOSTRING_USER_INNER = 0x00000008,
98:     CIT_CATCH_GET_CHILD = 0x00000010,
99:     CIT_FULL_CACHE = 0x00000100,
100:     CIT_PUBLIC = 0x000000FFFF,
101:     /* private */
102:     CIT_VALID = 0x00010000,
103:     CIT_HAS_CHILDREN = 0x00020000
104: };
105:
106: enum {
107:     /* public */
108:     REGIT_USER_KEY = 0x00000001,
109:     REGIT_INVERTED = 0x00000002
110: };
111:
112: typedef enum {
113:     REGIT_MODE_MATCH,
114:     REGIT_MODE_GET_MATCH,
115:     REGIT_MODE_ALL_MATCHES,
116:     REGIT_MODE_SPLIT,
117:     REGIT_MODE_REPLACE,
118:     REGIT_MODE_MAX
119: } regex_mode;
120:
121: typedef struct _spl_chfilter_it_intern {
122:     zend_fcall_info fci;
123:     zend_fcall_info_cache fcci;
124:     zend_object *obj;
125: } _spl_chfilter_it_intern;
126:
127: typedef struct _spl_dual_it_object {
128:     struct {
129:         zval *obj;
130:         zend_class_entry *ce;
131:         zend_object *obj;
132:         zend_object_iterator *iterator;
133:     } common;
134:     struct {
135:         zval *data;
136:         zval *key;
137:         zend_long pos;
138:     } current;
139:     dual_it_type dit_type;
140:     union {
141:         struct {
142:             zend_long offset;
143:             zend_long count;
144:         } limit;
145:         struct {
146:             zend_long flags; /* CIT_* */
147:             zval *scr;
148:             zval *children;
149:             zval *scache;
150:         } caching;
151:         struct {
152:             zval *array;
153:             zend_object_iterator *iterator;
154:         } append;
155: #if HAVE_PCRE
156:         struct {
157:             zend_long flags;
158:             zend_long preg_flags;
159:             pcre_cache_entry *pcre;
160:             zend_string *regex;
161:             regex_mode mode;
162:             int use_flags;
163:         } regex;
164: #endif
165:     } _spl_chfilter_it_intern *chfilter;
166: } _spl_dual_it_object;
167:
168: static inline spl_dual_it_object *spl_dual_it_from_obj(zend_object *obj) {
169:     return (spl_dual_it_object *) ((char *) (obj) - XOFFSETOF(spl_dual_it_object, std));
170: }
171:
172: /* }}} */
173:
174: #define SPL_DUAL_IT_P(xv) spl_dual_it_from_obj((zend_object *) (xv))
175:
176: typedef int (*spl_iterator_apply_func_t)(zend_object_iterator *iter, void *puser);
177:
178: PHPAPI int spl_iterator_apply(zval *obj, spl_iterator_apply_func_t apply_func, void *puser);
179:
180: #endif /* SPL_ITERATORS_H */
181:
182: /*
183:  * Local Variables:
184:  * * eval-offsets: 4
185:  * * tab-width: 4
186:  * * End:
187:  * * vim600: fdm=marker
188:  * * vim: noet sw=4 ts=4
189:  */
```

189: /*


```

1: 1:  PHP Version 7.0.0
2: 2:  Copyright (c) 1997-2018 The PHP Group
3: 3:  This source file is subject to version 3.01 of the PHP license,
4: 4:  that is bundled with this package in the file LICENSE, and is
5: 5:  available through the world-wide-web at the following url:
6: 6:  http://www.php.net/license/3_01.txt
7: 7:  If you did not receive a copy of the PHP license and are unable to
8: 8:  obtain it through the world-wide-web, please send a note to
9: 9:  license@php.net so we can mail you a copy immediately.
10: 10:  Authors: Marcus Ruescher <chelly@php.net>
11: 11:
12: 12:
13: 13:
14: 14:
15: 15:
16: 16:
17: 17:
18: 18:
19: 19:
20: 20:
21: 21: #ifndef HAVE_CONFIG_H
22: 22: #include "config.h"
23: 23: #endif
24: 24:
25: 25: #include "php.h"
26: 26: #include "php_ini.h"
27: 27: #include "ext/standard/info.h"
28: 28: #include "ext/standard/php_var.h"
29: 29: #include "zend_smart_str.h"
30: 30: #include "zend_interfaces.h"
31: 31: #include "zend_exceptions.h"
32: 32:
33: 33: #include "php_api.h"
34: 34: #include "api_functions.h"
35: 35: #include "api_globals.h"
36: 36: #include "api_iterators.h"
37: 37: #include "api_array.h"
38: 38: #include "api_exceptions.h"
39: 39:
40: 40: zend_object_handlers spl_handler_ArrayObject;
41: 41: PHPAPI zend_class_entry *spl_array_array_iterator;
42: 42:
43: 43: zend_object_handlers spl_handler_ArrayIterator;
44: 44: PHPAPI zend_class_entry *spl_array_array_iterator;
45: 45: PHPAPI zend_class_entry *spl_array_recurse_array_iterator;
46: 46:
47: 47: #define SPL_ARRAY_STD_PROPS_LIST 0x00000001
48: 48: #define SPL_ARRAY_ARRAY_AS_PROPS 0x00000002
49: 49: #define SPL_ARRAY_CHILD_ARRAYS_ONLY 0x00000004
50: 50: #define SPL_ARRAY_OVERLOADED_NEXT 0x00010000
51: 51: #define SPL_ARRAY_OVERLOADED_VALID 0x00020000
52: 52: #define SPL_ARRAY_OVERLOADED_KEY 0x00040000
53: 53: #define SPL_ARRAY_OVERLOADED_CURRENT 0x00080000
54: 54: #define SPL_ARRAY_OVERLOADED_NEXT 0x00100000
55: 55: #define SPL_ARRAY_IS_SELF 0x01000000
56: 56: #define SPL_ARRAY_USE_OTHER 0x02000000
57: 57: #define SPL_ARRAY_INT_MASK 0xFF000000
58: 58: #define SPL_ARRAY_CLONE_MASK 0x100FFFFF
59: 59:
60: 60: #define SPL_ARRAY_METHOD_NO_ARG 0
61: 61: #define SPL_ARRAY_METHOD_USE_ARG 1
62: 62: #define SPL_ARRAY_METHOD_USE_ARG 2
63: 63:
64: 64: typedef struct _spl_array_object {
65: 65:     zval array;
66: 66:     uint32_t ht_iter;
67: 67:     int flags;
68: 68:     unsigned char *obj_properties;
69: 69:     zend_function *fptr_offset_get;
70: 70:     zend_function *fptr_offset_set;
71: 71:     zend_function *fptr_offset_key;
72: 72:     zend_function *fptr_offset_del;
73: 73:     zend_function *fptr_count;
74: 74:     zend_class_entry *ce_get_iterator;
75: 75:     zend_object std;
76: 76:     spl_array_object;
77: 77:
78: 78:     static inline spl_array_object *spl_array_from_obj(zend_object *obj) { /* {{{ */
79: 79:         return (spl_array_object *) (char *) (obj) - XOffsetOf(spl_array_object, std);
80: 80:     }
81: 81:     /* }}} */
82: 82:
83: 83: #define Z_SPARRAY_P(rv) spl_array_from_obj(Z_OBJ_P((rv)))
84: 84:
85: 85:     static inline HashTable **spl_array_get_hash_table_ptr(spl_array_object *intern) { /* {{{ */
86: 86:         /*??? TODO: Delay duplication for arrays; only duplicate for write operations
87: 87:         if (intern->var_flags & SPL_ARRAY_IS_SELF) {
88: 88:             if (intern->std.properties) {
89: 89:                 rebuild_obj_properties(intern->std);
90: 90:             }
91: 91:             return intern->std.properties;
92: 92:         } else if (intern->var_flags & SPL_ARRAY_USE_OTHER) {
93: 93:             spl_array_object *other = Z_SPARRAY_P(intern->array);
94: 94:             return spl_array_get_hash_table_ptr(other);
95: 95:         } else if (Z_TYPE(intern->array) == IS_ARRAY) {
96: 96:             return Z_ARRVAL(intern->array);
97: 97:         } else {
98: 98:             zend_object *obj = Z_OBJ(intern->array);
99: 99:             if (obj->properties) {
100: 100:                 rebuild_obj_properties(obj);
101: 101:             } else if (GC_REFCOUNT(obj->properties) > 1) {
102: 102:                 IF EXPECTED (! GC_FLAGS(obj->properties) & IS_ARRAY_IMMUTABLE);
103: 103:                 GC_REFCOUNT(obj->properties);
104: 104:             }
105: 105:             obj->properties = zend_array_dup(obj->properties);
106: 106:             return obj->properties;
107: 107:         }
108: 108:     }
109: 109:     /* }}} */
110: 110:     /* {{{ */
111: 111:     static inline HashTable *spl_array_get_hash_table(spl_array_object *intern) { /* {{{ */
112: 112:         return *spl_array_get_hash_table_ptr(intern);
113: 113:     }
114: 114:     /* }}} */
115: 115:     /* {{{ */
116: 116:     static inline void spl_array_replace_hash_table(spl_array_object *intern, HashTable *ht) { /* {{{ */
117: 117:         HashTable **ht_ptr = spl_array_get_hash_table_ptr(intern);
118: 118:         zend_array_destroy(*ht_ptr);
119: 119:         *ht_ptr = ht;
120: 120:     }
121: 121:     /* }}} */
122: 122:     /* {{{ */
123: 123:     static inline zend_bool spl_array_is_object(spl_array_object *intern) { /* {{{ */
124: 124:         while (intern->var_flags & SPL_ARRAY_USE_OTHER) {
125: 125:             intern = Z_SPARRAY_P(intern->array);
126: 126:         }
127: 127:         return (intern->var_flags & SPL_ARRAY_IS_SELF) || (Z_TYPE(intern->array) == IS_OBJECT);
128: 128:     }
129: 129:     /* }}} */
130: 130:     /* {{{ */
131: 131:     static inline spl_array_skip_protected(spl_array_object *intern, HashTable *ht);
132: 132:     static inline void spl_array_create_ht_iter(HashTable *ht, spl_array_object *intern) { /* {{{ */
133: 133:         intern->ht_iter = zend_hash_iterator_add(ht, ht->internalPointer);
134: 134:         zend_hash_internal_pointer_reset_ex(ht, &ht->internalPointer);
135: 135:         spl_array_skip_protected(intern, ht);
136: 136:     }
137: 137:     /* }}} */
138: 138:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
139: 139:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
140: 140:             spl_array_create_ht_iter(ht, intern);
141: 141:         }
142: 142:         return &ht->internalPointer;
143: 143:     }
144: 144:     /* }}} */
145: 145:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
146: 146:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
147: 147:             spl_array_create_ht_iter(ht, intern);
148: 148:         }
149: 149:         return &ht->internalPointer;
150: 150:     }
151: 151:     /* {{{ */
152: 152:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
153: 153:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
154: 154:             spl_array_create_ht_iter(ht, intern);
155: 155:         }
156: 156:         return &ht->internalPointer;
157: 157:     }
158: 158:     /* }}} */
159: 159:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
160: 160:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
161: 161:             spl_array_create_ht_iter(ht, intern);
162: 162:         }
163: 163:         return &ht->internalPointer;
164: 164:     }
165: 165:     /* }}} */
166: 166:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
167: 167:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
168: 168:             spl_array_create_ht_iter(ht, intern);
169: 169:         }
170: 170:         return &ht->internalPointer;
171: 171:     }
172: 172:     /* }}} */
173: 173:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
174: 174:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
175: 175:             spl_array_create_ht_iter(ht, intern);
176: 176:         }
177: 177:         return &ht->internalPointer;
178: 178:     }
179: 179:     /* }}} */
180: 180:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
181: 181:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
182: 182:             spl_array_create_ht_iter(ht, intern);
183: 183:         }
184: 184:         return &ht->internalPointer;
185: 185:     }
186: 186:     /* }}} */
187: 187:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
188: 188:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
189: 189:             spl_array_create_ht_iter(ht, intern);
190: 189:         }
191: 190:         return &ht->internalPointer;
192: 191:     }
193: 191:     /* }}} */
194: 191:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
195: 195:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
196: 196:             spl_array_create_ht_iter(ht, intern);
197: 196:         }
198: 196:         return &ht->internalPointer;
199: 196:     }
200: 196:     /* }}} */
201: 196:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
202: 202:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
203: 203:             spl_array_create_ht_iter(ht, intern);
204: 203:         }
205: 203:         return &ht->internalPointer;
206: 203:     }
207: 203:     /* }}} */
208: 203:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
209: 209:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
210: 209:             spl_array_create_ht_iter(ht, intern);
211: 209:         }
212: 209:         return &ht->internalPointer;
213: 209:     }
214: 209:     /* }}} */
215: 209:     static inline void spl_array_free_storage(spl_array_object *intern) { /* {{{ */
216: 216:         if (UNEXPECTED(intern->ht_iter == (uint32_t)-1)) {
217: 216:             spl
```

[illegible]

```

753: if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "es", &index, &value) == FAILURE) {
754:     return;
755: }
756: spl_array_write_dimension_ex(0, getThis(), index, value);
757: } /* }}} */
758:
759: void spl_array_iterator_append(zval *object, zval *append_value) /* {{{ */
760: {
761:     spl_array_object *intern = Z_SPLARRAY_P(object);
762:     HashTable *aht = spl_array_get_hash_table(intern);
763:
764:     if (!aht) {
765:         php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
766:         return;
767:     }
768:
769:     if (spl_array_is_object(intern)) {
770:         zend_throw_error(NULL, "Cannot append properties to objects, use %s::offsetSet() instead", ZSTR_VAL(Z_OBJCE_P(object)->name));
771:         return;
772:     }
773:
774:     spl_array_write_dimension(object, NULL, append_value);
775: } /* }}} */
776:
777: /* {{{ proto void ArrayObject::append(mixed $value)
778:    proto void ArrayIterator::append(mixed $value)
779:    Appends the value (cannot be called for objects). */
780: #define SPL_METHOD(Array, append)
781:
782: zval *value;
783:
784: if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s", &value) == FAILURE) {
785:     return;
786: }
787: spl_array_iterator_append(getThis(), value);
788: } /* }}} */
789:
790: /* {{{ proto void ArrayObject::offsetUnset(mixed $index)
791:    proto void ArrayIterator::offsetUnset(mixed $index)
792:    Unsets the value at the specified $index. */
793: #define SPL_METHOD(Array, offsetUnset)
794:
795: zval *index;
796: if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s", &index) == FAILURE) {
797:     return;
798: }
799: spl_array_unset_dimension_ex(0, getThis(), index);
800: } /* }}} */
801:
802: /* {{{ proto array ArrayObject::getArrayCopy()
803:    proto array ArrayIterator::getArrayCopy()
804:    Returns a copy of the contained array */
805: #define SPL_METHOD(Array, getArrayCopy)
806:
807: zval *object = getThis();
808: spl_array_object *intern = Z_SPLARRAY_P(object);
809:
810: RETURN_ARRAY(zend_array_dup(spl_array_get_hash_table(intern)));
811: } /* }}} */
812:
813: static HashTable *spl_array_get_properties(zval *object) /* {{{ */
814: {
815:     spl_array_object *intern = Z_SPLARRAY_P(object);
816:
817:     if ((intern->var_flags & SPL_ARRAY_STD_PROP_LIST) &
818:         !((intern->std.properties) &
819:           rebuild_object_properties(&intern->std))) {
820:         return intern->std.properties;
821:     }
822: }
823:
824: return spl_array_get_hash_table(intern);
825: } /* }}} */
826:
827: static HashTable * spl_array_get_debug_info(zval *obj, int *is_temp) /* {{{ */
828: {
829:     zval *storage;
830:     zend_string *name;
831:     zend_string *key;
832:     spl_array_object *intern = Z_SPLARRAY_P(obj);
833:
834:     if ((intern->std.properties) &
835:         rebuild_object_properties(&intern->std)) {
836:     }
837:
838:     if ((intern->var_flags & SPL_ARRAY_IS_SELF) &
839:         *is_temp == 0) {
840:         return intern->std.properties;
841:     } else {
842:         HashTable *debug_info;
843:         *is_temp = 1;
844:
845:         debug_info = zend_new_array(zend_hash_num_elements(intern->std.properties) + 1);
846:         zend_hash_copy(debug_info, intern->std.properties, (copy_ctor_func_t) zval_add_ref);
847:
848:         storage = &intern->array;
849:         Z_TRY_ADDREF_P(storage);
850:
851:         base = Z_OBJ_HT_P(obj) == spl_handler_ArrayIterator
852:             ? spl_obj_ArrayIterator : spl_obj_ArrayObject;
853:         name = spl_get_private_prop_name(base, "storage", &is_temp);
854:         zend_symtable_update(debug_info, name, storage);
855:         zend_string_release(name);
856:
857:         return debug_info;
858:     }
859: }
860: } /* }}} */
861:
862: static HashTable *spl_array_get_gc(zval *obj, zval **gc_data, int *gc_data_count) /* {{{ */
863: {
864:     spl_array_object *intern = Z_SPLARRAY_P(obj);
865:     *gc_data = &intern->array;
866:     *gc_data_count = 1;
867:     return zend_std_get_properties(obj);
868: } /* }}} */
869:
870: static zval *spl_array_read_property(zval *object, zval *member, int type, void **cache_slot, zval **rv) /* {{{ */
871: {
872:     spl_array_object *intern = Z_SPLARRAY_P(object);
873:
874:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0) {
875:         if (!std_object_handlers.has_property(object, member, Z_NULL)) {
876:             return spl_array_read_dimension(object, member, type, rv);
877:         }
878:     }
879:     return std_object_handlers.read_property(object, member, type, cache_slot, rv);
880: } /* }}} */
881:
882: static void spl_array_write_property(zval *object, zval *member, zval *value, void **cache_slot) /* {{{ */
883: {
884:     spl_array_object *intern = Z_SPLARRAY_P(object);
885:
886:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0) {
887:         if (!std_object_handlers.has_property(object, member, Z_NULL)) {
888:             spl_array_write_dimension(object, member, value);
889:             return;
890:         }
891:     }
892:     std_object_handlers.write_property(object, member, value, cache_slot);
893: } /* }}} */
894:
895: static zval *spl_array_get_property_ptr_ptr(zval *object, zval *member, int type, void **cache_slot) /* {{{ */
896: {
897:     spl_array_object *intern = Z_SPLARRAY_P(object);
898:
899:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0) {
900:         if (!std_object_handlers.has_property(object, member, Z_NULL)) {
901:             /* If object has offset() or offsetUnset(), then fallback to read_property,
902              * which will call offsetGet(). */
903:             if ((intern->flags & SPL_OFFSET_GET) &
904:                 !std_object_handlers.has_property(object, member, has_offset_exists, cache_slot)) {
905:                 return NULL;
906:             }
907:             return spl_array_get_dimension_ptr(1, intern, member, type);
908:         }
909:     }
910:     return std_object_handlers.get_property_ptr_ptr(object, member, type, cache_slot);
911: } /* }}} */
912:
913: static int spl_array_has_property(zval *object, zval *member, int has_offset_exists, void **cache_slot) /* {{{ */
914: {
915:     spl_array_object *intern = Z_SPLARRAY_P(object);
916:
917:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0) {
918:         if (!std_object_handlers.has_property(object, member, Z_NULL)) {
919:             return spl_array_has_dimension(object, member, has_offset_exists, cache_slot);
920:         }
921:     }
922:     return std_object_handlers.has_property(object, member, has_offset_exists, cache_slot);
923: } /* }}} */
924:
925: static void spl_array_unset_property(zval *object, zval *member, void **cache_slot) /* {{{ */
926: {
927:     spl_array_object *intern = Z_SPLARRAY_P(object);
928:
929:     if ((intern->var_flags & SPL_ARRAY_ARRAY_AS_PROPS) != 0) {
930:         if (!std_object_handlers.has_property(object, member, Z_NULL)) {
931:             spl_array_unset_dimension(object, member);
932:             return;
933:         }
934:     }
935:     std_object_handlers.unset_property(object, member, cache_slot);
936: } /* }}} */
937:
938: static int spl_array_compare_objects(zval *o1, zval *o2) /* {{{ */
939: {
940:     HashTable *ht1,
941:               *ht2;
942:     spl_array_object *intern1,
943:                     *intern2;
944:     int result = 0;
945:
946:     intern1 = Z_SPLARRAY_P(o1);
947:     intern2 = Z_SPLARRAY_P(o2);
948:     ht1 = spl_array_get_hash_table(intern1);
949:     ht2 = spl_array_get_hash_table(intern2);
950:
951:     result = zend_compare_symbol_tables(ht1, ht2);
952:     /* If we just compared std.properties, don't do it again */
953:     if (result == 0 &&
954:         !((ht1 == intern1->std.properties && ht2 == intern2->std.properties) &
955:           result == std_object_handlers.compare_objects(o1, o2))) {
956:         return result;
957:     }
958:     /* }}} */
959:
960:     static int spl_array_skip_protected(spl_array_object *intern, HashTable *aht) /* {{{ */
961:     {
962:         zend_string *attr_key;
963:         zend_ulong num_key;
964:         zval *data;
965:
966:         if (spl_array_is_object(intern)) {
967:             uint32_t *pos_ptr = spl_array_get_pos_ptr(aht, intern);
968:
969:             do {
970:                 if (zend_hash_get_current_key_ex(aht, attr_key, num_key, pos_ptr) == HASH_KEY_IS_STRING) {
971:                     data = zend_hash_get_current_data_ex(aht, pos_ptr);
972:                     if (data && Z_TYPE_P(data) == IS_INDIRECT &&
973:                         Z_TYPE_P(*data) == Z_INDIRECT_P(data) == IS_UNDEF) {
974:                         /* skip */
975:                     } else if (!ZSTR_LEN(string_key) || ZSTR_VAL(string_key)[0]) {
976:                         return SUCCESS;
977:                     }
978:                 } else {
979:                     return SUCCESS;
980:                 }
981:                 if (zend_hash_has_more_elements_ex(aht, pos_ptr) != SUCCESS) {
982:                     return FAILURE;
983:                 }
984:                 zend_hash_move_forward_ex(aht, pos_ptr);
985:                 while (!);
986:             } while (!);
987:             return FAILURE;
988:         }
989:         /* }}} */
990:
991:         static int spl_array_next(spl_array_object *intern, HashTable *aht) /* {{{ */
992:         {
993:             uint32_t *pos_ptr = spl_array_get_pos_ptr(aht, intern);
994:
995:             zend_hash_move_forward_ex(aht, pos_ptr);
996:             if (spl_array_is_object(intern)) {
997:                 return spl_array_skip_protected(intern, aht);
998:             }
999:             return zend_hash_has_more_elements_ex(aht, pos_ptr);
1000:         }
1001:         /* }}} */
1002:
1003:         static void spl_array_it_dtor(zend_object_iterator *iter) /* {{{ */
1004:         {
1005:             zend_user_it_invalidate_current(iter);
1006:             zval_ptr_dtor(&iter->data);
1007:         }
1008:         /* }}} */
1009:
1010:         static int spl_array_it_valid(zend_object_iterator *iter) /* {{{ */
1011:         {
1012:             spl_array_object *object = Z_SPLARRAY_P(iter->data);
1013:             HashTable *aht = spl_array_get_hash_table(object);
1014:
1015:             if (object->var_flags & SPL_ARRAY_OVERLOADED_VALID) {
1016:                 return zend_user_it_valid(iter);
1017:             }
1018:             if (spl_array_object_verify_pos_ex(object, aht, "ArrayIterator::valid()") == FAILURE) {
1019:                 return FAILURE;
1020:             }
1021:             return zend_hash_has_more_elements_ex(aht, spl_array_get_pos_ptr(aht, object));
1022:         }
1023:         /* }}} */
1024:
1025:         static zval *spl_array_it_get_current_data(zend_object_iterator *iter) /* {{{ */
1026:         {
1027:             spl_array_object *object = Z_SPLARRAY_P(iter->data);
1028:             HashTable *aht = spl_array_get_hash_table(object);
1029:
1030:             if (object->var_flags & SPL_ARRAY_OVERLOADED_CURRENT) {
1031:                 return zend_user_it_get_current_data(iter);
1032:             }
1033:             if (Z_TYPE_P(data) == IS_INDIRECT &&
1034:                 data = Z_INDIRECT_P(data)) {
1035:                 return data;
1036:             }
1037:             return NULL;
1038:         }
1039:         /* }}} */
1040:
1041:         static void spl_array_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
1042:         {
1043:             spl_array_object *object = Z_SPLARRAY_P(iter->data);
1044:             HashTable *aht = spl_array_get_hash_table(object);
1045:
1046:             if (object->var_flags & SPL_ARRAY_OVERLOADED_KEY) {
1047:                 zend_user_it_get_current_key(iter, key);
1048:             }
1049:             if (spl_array_object_verify_pos_ex(object, aht, "ArrayIterator::current()") == FAILURE) {
1050:                 return NULL;
1051:             }
1052:             zend_hash_get_current_key_zval_ex(aht, key, spl_array_get_pos_ptr(aht, object));
1053:         }
1054:         /* }}} */
1055:
1056:         static void spl_array_it_move_forward(zend_object_iterator *iter) /* {{{ */
1057:         {
1058:             spl_array_object *object = Z_SPLARRAY_P(iter->data);
1059:             HashTable *aht = spl_array_get_hash_table(object);
1060:
1061:             if (object->var_flags & SPL_ARRAY_OVERLOADED_NEXT) {
1062:                 zend_user_it_move_forward(iter);
1063:             }
1064:             if (Z_TYPE_P(data) == IS_INDIRECT &&
1065:                 data = Z_INDIRECT_P(data)) {
1066:                 return data;
1067:             }
1068:             return NULL;
1069:         }
1070:         /* }}} */
1071:
1072:         static void spl_array_it_rewind(spl_array_object *intern) /* {{{ */
1073:         {
1074:             if (!aht) {
1075:                 php_error_docref(NULL, E_NOTICE, "ArrayIterator::rewind(): Array was modified outside object and is no longer an array");
1076:                 return;
1077:             }
1078:             spl_array_next_ex(object, aht);
1079:         }
1080:         /* }}} */
1081:
1082:         static void spl_array_it_rewind(spl_array_object *intern) /* {{{ */
1083:         {
1084:             HashTable *aht = spl_array_get_hash_table(intern);
1085:
1086:             if (!aht) {
1087:                 php_error_docref(NULL, E_NOTICE, "ArrayIterator::rewind(): Array was modified outside object and is no longer an array");
1088:                 return;
1089:             }
1090:             if (intern->std.iter == (uint32_t)-1) {
1091:                 spl_array_get_pos_ptr(aht, intern);
1092:             }
1093:             if (aht) {
1094:                 zend_hash_internal_pointer_reset_ex(aht, spl_array_get_pos_ptr(aht, intern));
1095:                 spl_array_skip_protected(intern, aht);
1096:             }
1097:         }
1098:         /* }}} */
1099:
1100:         static void spl_array_it_rewind(spl_array_object *intern) /* {{{ */
1101:         {
1102:             spl_array_object *object = Z_SPLARRAY_P(iter->data);
1103:             HashTable *aht = spl_array_get_hash_table(object);
1104:
1105:             if (object->var_flags & SPL_ARRAY_OVERLOADED_REWIND) {
1106:                 zend_user_it_rewind(iter);
1107:             }
1108:             if (Z_TYPE_P(data) == IS_INDIRECT &&
1109:                 data = Z_INDIRECT_P(data)) {
1110:                 return data;
1111:             }
1112:             return NULL;
1113:         }
1114:         /* }}} */
1115:
1116:         static void spl_array_it_rewind(spl_array_object *intern, zval *array, zend_ulong ar_flags, int just_array) /* {{{ */
1117:         {
1118:             if (Z_TYPE_P(array) == IS_ARRAY) {
1119:                 zval_ptr_dtor(&intern->array);
1120:                 if (Z_TYPE_P(array) == IS_ARRAY) {
1121:                     ZVAL_COPY(&intern->array, array);
1122:                 }
1123:                 //??? TODO: try to avoid array duplication
1124:                 ZVAL_ARR(&intern->array, zend_array_dup(Z_ARR_P(array)));
1125:             }
1126:         }
1127:         /* }}} */

```

```

1:         if (Z_OBJ_HT_P(array) == spl_handler_ArrayObject) {
2:             zval_get_ztor(intern->array);
3:             if (!zval_array(array))
4:                 spl_array_object_throw = 2_SPLARRAY_P(array);
5:             ar_flags = other->ar_flags & "SPL_ARRAY_INT_MASK";
6:         }
7:
8:         if (Z_OBJ_P(object) == Z_OBJ_P(array)) {
9:             ar_flags = SPL_ARRAY_IS_SELF;
10:            ZVAL_UNDEF(&intern->array);
11:        } else {
12:            ar_flags = SPL_ARRAY_USE_OTHER;
13:            ZVAL_COPY(&intern->array, array);
14:        }
15:    } else {
16:        zend_object_get_properties_t_handler = Z_OBJ_HANDLER_P(array, get_properties);
17:        if (handler != std_object_handlers.get_properties) {
18:            zend_throw_exception_ex(spl_ce_invalidArgumentException, 0,
19:                "Overloaded object of type %s is not compatible with %s",
20:                ESTR_VAL(Z_OBJCE_P(array)->name), ESTR_VAL(intern->std.ce->name));
21:            return;
22:        }
23:    }
24:
25:    intern->ar_flags = "SPL_ARRAY_IS_SELF" & "SPL_ARRAY_USE_OTHER";
26:    intern->ar_flags |= ar_flags;
27:    intern->znt_intern = (uint32_t)-1;
28:
29:    /* }}} */
30:
31:    /* iterator handler table */
32:    static const zend_object_iterator_funcs spl_array_it_funcs = {
33:        spl_array_it_dtor,
34:        spl_array_it_valid,
35:        spl_array_it_get_current_data,
36:        spl_array_it_get_next_key,
37:        spl_array_it_move_forward,
38:        spl_array_it_rewind,
39:        NULL
40:    };
41:
42:    zend_object_iterator *spl_array_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
43:    {
44:        zend_user_iterator *iterator;
45:        zend_splarray_p(object);
46:
47:        if (by_ref & (spl_array_object_ar_flags & SPL_ARRAY_OVERLOADED_CURRENT)) {
48:            zend_throw_exception(spl_ce_runtime_exception, "No iterator cannot be used with foreach by reference", 0);
49:            return NULL;
50:        }
51:
52:        iterator = emalloc(sizeof(zend_user_iterator));
53:
54:        zend_iterator_init(iterator->it);
55:
56:        ZVAL_COPY(&iterator->it.data, object);
57:        iterator->it.funcs = spl_array_it_funcs;
58:        iterator->ce = ce;
59:        ZVAL_UNDEF(iterator->value);
60:
61:        return iterator->it;
62:    }
63:
64:    /* }}} */
65:
66:    /* {{{ proto void ArrayObject::__construct([array|object ar = array() [, int flags = 0 [, string iterator_class]])
67:     * Constructs a new array iterator from an array or object. */
68:    SPL_METHOD(Array, __construct)
69:    {
70:        zval *object = &this();
71:        spl_array_object_intern;
72:        zval *array;
73:        zend_long ar_flags = 0;
74:        zend_class_entry *ce_get_iterator = spl_ce_iterator;
75:
76:        if (ZEND_NUM_ARGS() == 0) {
77:            return; /* nothing to do */
78:        }
79:
80:        if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "e|l", &array, &ar_flags, &ce_get_iterator) == FAILURE) {
81:            return;
82:        }
83:
84:        intern = Z_SPLARRAY_P(object);
85:
86:        if (ZEND_NUM_ARGS() > 2) {
87:            intern->ce_get_iterator = ce_get_iterator;
88:
89:            ar_flags |= "SPL_ARRAY_INT_MASK";
90:
91:            spl_array_set_array(object, intern, array, ar_flags, ZEND_NUM_ARGS() == 1);
92:        }
93:    }
94:
95:    /* }}} */
96:
97:    /* {{{ proto void ArrayIterator::__construct([array|object ar = array() [, int flags = 0])
98:     * Constructs a new array iterator from an array or object. */
99:    SPL_METHOD(ArrayIterator, __construct)
100:    {
101:        zval *object = &this();
102:        spl_array_object_intern;
103:        zval *array;
104:        zend_long ar_flags = 0;
105:
106:        if (ZEND_NUM_ARGS() == 0) {
107:            return; /* nothing to do */
108:        }
109:
110:        if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "e|l", &array, &ar_flags) == FAILURE) {
111:            return;
112:        }
113:
114:        intern = Z_SPLARRAY_P(object);
115:
116:        ar_flags |= "SPL_ARRAY_INT_MASK";
117:
118:        spl_array_get_array(object, intern, array, ar_flags, ZEND_NUM_ARGS() == 1);
119:    }
120:
121:    /* }}} */
122:
123:    /* {{{ proto void ArrayObject::setIteratorClass(string iterator_class)
124:     * Set the class used in getIterator. */
125:    SPL_METHOD(Array, setIteratorClass)
126:    {
127:        zval *object = &this();
128:        spl_array_object_intern = Z_SPLARRAY_P(object);
129:        zval *array;
130:        zend_class_entry *ce_get_iterator = spl_ce_iterator;
131:
132:        ZEND_PARSE_PARAMETERS_START(1, 1)
133:            Z_PARAM_CLASS(ce_get_iterator)
134:        ZEND_PARSE_PARAMETERS_END();
135:
136:        intern->ce_get_iterator = ce_get_iterator;
137:    }
138:
139:    /* }}} */
140:
141:    /* {{{ proto string ArrayObject::getIteratorClass()
142:     * Get the class used in getIterator. */
143:    SPL_METHOD(Array, getIteratorClass)
144:    {
145:        zval *object = &this();
146:        spl_array_object_intern = Z_SPLARRAY_P(object);
147:
148:        if (zend_parse_parameters_none() == FAILURE) {
149:            return;
150:        }
151:
152:        zend_string_offset(intern->ce_get_iterator->name);
153:        RETURN_STR(intern->ce_get_iterator->name);
154:    }
155:
156:    /* }}} */
157:
158:    /* {{{ proto int ArrayObject::getFlags()
159:     * Get flags */
160:    SPL_METHOD(Array, getFlags)
161:    {
162:        zval *object = &this();
163:        spl_array_object_intern = Z_SPLARRAY_P(object);
164:
165:        if (zend_parse_parameters_none() == FAILURE) {
166:            return;
167:        }
168:
169:        RETURN_LONG(intern->ar_flags & "SPL_ARRAY_INT_MASK");
170:    }
171:
172:    /* }}} */
173:
174:    /* {{{ proto void ArrayObject::setFlags(int flags)
175:     * Set flags */
176:    SPL_METHOD(Array, setFlags)
177:    {
178:        zval *object = &this();
179:        spl_array_object_intern = Z_SPLARRAY_P(object);
180:        zend_long ar_flags = 0;
181:
182:        if (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &ar_flags) == FAILURE) {
183:            return;
184:        }
185:
186:        intern->ar_flags = (intern->ar_flags & SPL_ARRAY_INT_MASK) | (ar_flags & "SPL_ARRAY_INT_MASK");
187:    }
188:
189:    /* }}} */
190:
191:    /* {{{ proto Array|Object ArrayObject::exchangeArray(Array|Object ar = array())
192:     * Replace the referenced array or object with a new one and return the old one (right now copy - to be changed)
193:     * Array, exchangeArray
194:     */

```

```

3117:     sval *object = getThis(), *array;
3118:     spl_array_object *intern = _S_PLARRAY_P(object);
3119: }
3120:
3121: zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "a", &array) == FAILURE {
3122:     return;
3123: }
3124:
3125: if (intern->mapCount > 0) {
3126:     zend_error(E_NOTICE, "Modification of ArrayObject during sorting is prohibited");
3127: }
3128: return;
3129:
3130: RETVAL_ARR(zend_array_dup(spl_array_get_hash_table(intern)));
3131: spl_array_set_array(object, intern, array, 0, 1);
3132: }
3133: }
3134:
3135: /* {{{ proto ArrayIterator ArrayObject::getIterator()
3136:  * Create a new iterator from a ArrayObject instance */
3137: SPL_METHOD(Array, getIterator)
3138: {
3139:     sval *object = getThis();
3140:     spl_array_object *intern = _S_PLARRAY_P(object);
3141:     HashTable *ah = spl_array_get_hash_table(intern);
3142:
3143:     if (zend_parse_parameters_none() == FAILURE) {
3144:         return;
3145:     }
3146:
3147:     if (finfo) {
3148:         php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
3149:         return;
3150:     }
3151:
3152:     ZVAL_OBJ(&return_value, spl_array_object_new_as(intern->obj_get_iterator(), object, 0));
3153: }
3154:
3155: /* {{{ proto void ArrayIterator::rewind()
3156:  * Rewind array back to the start */
3157: SPL_METHOD(Array, rewind)
3158: {
3159:     sval *object = getThis();
3160:     spl_array_object *intern = _S_PLARRAY_P(object);
3161:
3162:     if (zend_parse_parameters_none() == FAILURE) {
3163:         return;
3164:     }
3165:
3166:     spl_array_rewind(intern);
3167: }
3168: }
3169:
3170: /* {{{ proto void ArrayIterator::seek(int $position)
3171:  * Seek to position. */
3172: SPL_METHOD(Array, seek)
3173: {
3174:     zend_long opos, position;
3175:     sval *object = getThis();
3176:     spl_array_object *intern = _S_PLARRAY_P(object);
3177:     HashTable *ah = spl_array_get_hash_table(intern);
3178:     int result;
3179:
3180:     if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "i", &position) == FAILURE) {
3181:         return;
3182:     }
3183:
3184:     if (finfo) {
3185:         php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
3186:         return;
3187:     }
3188:
3189:     opos = position;
3190:
3191:     if (position == 0) { /* negative values are not supported */
3192:         spl_array_rewind(intern);
3193:         result = SUCCESS;
3194:     }
3195:
3196:     while (position-- > 0 && (result = spl_array_next(intern)) == SUCCESS;
3197:     if (result == SUCCESS && zend_hash_has_more_elements_ex(ah, spl_array_get_pos_ptr(ah, intern)) == SUCCESS) {
3198:         return; /* ok */
3199:     }
3200:
3201:     zend_throw_exception_ex(spl_ce_OutOfBoundsException, 0, "Seek position " ZEND_LONG_FMT " is out of range", opos);
3202: }
3203: }
3204:
3205: static int spl_array_object_count_elements_helper(spl_array_object *intern, zend_long *count) /* {{{ */
3206: {
3207:     HashTable *ah = spl_array_get_hash_table(intern);
3208:     HashPosition pos, *pos_ptr;
3209:
3210:     if (finfo) {
3211:         php_error_docref(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
3212:         *count = 0;
3213:         return FAILURE;
3214:     }
3215:
3216:     if (spl_array_is_object(intern)) {
3217:         /* we need to store the 'pos' since we'll modify it in the functions
3218:          * we're going to call and which do not support 'pos' as parameter. */
3219:         pos_ptr = spl_array_get_pos_ptr(intern);
3220:         pos = *pos_ptr;
3221:         *count = 0;
3222:     }
3223:
3224:     spl_array_rewind(intern);
3225:     while (pos_ptr != HT_INVALID_IDX && spl_array_next(intern) == SUCCESS) {
3226:         (*count)++;
3227:     }
3228:
3229:     *pos_ptr = pos;
3230:     return SUCCESS;
3231: }
3232:
3233: *count = zend_hash_num_elements(ah);
3234: return SUCCESS;
3235: }
3236: }
3237:
3238: /* {{{ */
3239:
3240: int spl_array_object_count_elements(sval *object, zend_long *count) /* {{{ */
3241: {
3242:     spl_array_object *intern = _S_PLARRAY_P(object);
3243:
3244:     if (intern->fptr_count) {
3245:         sval rrv;
3246:         zend_call_method_with_0_params(object, intern->std_obj, intern->fptr_count, "count", &rv);
3247:         if (IS_TYPE_IV) {
3248:             *count = sval_get_long(&rv);
3249:             zend_throw_exception(spl_ce_BadMethodCallException, "Function expects one argument at most", 0);
3250:             return SUCCESS;
3251:         }
3252:         *count = 0;
3253:         return FAILURE;
3254:     }
3255:
3256:     return spl_array_object_count_elements_helper(intern, count);
3257: }
3258: }
3259:
3260: /* {{{ */
3261:
3262: /* {{{ proto int ArrayObject::count()
3263:  * Return the number of elements in the Iterator. */
3264: SPL_METHOD(Array, count)
3265: {
3266:     zend_long count;
3267:     spl_array_object *intern = _S_PLARRAY_P(getThis());
3268:
3269:     if (zend_parse_parameters_none() == FAILURE) {
3270:         return;
3271:     }
3272:
3273:     spl_array_object_count_elements_helper(intern, &count);
3274:
3275:     RETVAL_LONG(count);
3276: }
3277: }
3278:
3279: static void spl_array_method(INTERNAL_FUNCTION_PARAMETERS, char *fname, int fname_len, int use_arg) /* {{{ */
3280: {
3281:     spl_array_object *intern = _S_PLARRAY_P(getThis());
3282:     HashTable *ah = spl_array_get_hash_table(intern);
3283:     zend_function *f, *f2;
3284:     zend_string *f_name;
3285:
3286:     if (f_name = zend_string_init(fname, fname_len, 0)) {
3287:         ZVAL_STRING(f_name, f_name, 1);
3288:         ZVAL_UNDEF(f_name);
3289:         ZVAL_UNDEF(f_name);
3290:         ZVAL_UNDEF(f_name);
3291:         ZVAL_UNDEF(f_name);
3292:         ZVAL_UNDEF(f_name);
3293:         ZVAL_UNDEF(f_name);
3294:         ZVAL_UNDEF(f_name);
3295:         ZVAL_UNDEF(f_name);
3296:         ZVAL_UNDEF(f_name);
3297:         ZVAL_UNDEF(f_name);
3298:         ZVAL_UNDEF(f_name);
3299:         ZVAL_UNDEF(f_name);
3300:         ZVAL_UNDEF(f_name);
3301:         ZVAL_UNDEF(f_name);
3302:         ZVAL_UNDEF(f_name);
3303:         ZVAL_UNDEF(f_name);
3304:         ZVAL_UNDEF(f_name);
3305:         ZVAL_UNDEF(f_name);
3306:         ZVAL_UNDEF(f_name);
3307:         ZVAL_UNDEF(f_name);
3308:         ZVAL_UNDEF(f_name);
3309:         ZVAL_UNDEF(f_name);
3310:         ZVAL_UNDEF(f_name);
3311:         ZVAL_UNDEF(f_name);
3312:         ZVAL_UNDEF(f_name);
3313:         ZVAL_UNDEF(f_name);
3314:         ZVAL_UNDEF(f_name);
3315:         ZVAL_UNDEF(f_name);
3316:         ZVAL_UNDEF(f_name);
3317:         ZVAL_UNDEF(f_name);
3318:         ZVAL_UNDEF(f_name);
3319:         ZVAL_UNDEF(f_name);
3320:         ZVAL_UNDEF(f_name);
3321:         ZVAL_UNDEF(f_name);
3322:         ZVAL_UNDEF(f_name);
3323:         ZVAL_UNDEF(f_name);
3324:         ZVAL_UNDEF(f_name);
3325:         ZVAL_UNDEF(f_name);
3326:         ZVAL_UNDEF(f_name);
3327:         ZVAL_UNDEF(f_name);
3328:         ZVAL_UNDEF(f_name);
3329:         ZVAL_UNDEF(f_name);
3330:         ZVAL_UNDEF(f_name);
3331:         ZVAL_UNDEF(f_name);
3332:         ZVAL_UNDEF(f_name);
3333:         ZVAL_UNDEF(f_name);
3334:         ZVAL_UNDEF(f_name);
3335:         ZVAL_UNDEF(f_name);
3336:         ZVAL_UNDEF(f_name);
3337:         ZVAL_UNDEF(f_name);
3338:         ZVAL_UNDEF(f_name);
3339:         ZVAL_UNDEF(f_name);
3340:         ZVAL_UNDEF(f_name);
3341:         ZVAL_UNDEF(f_name);
3342:         ZVAL_UNDEF(f_name);
3343:         ZVAL_UNDEF(f_name);
3344:         ZVAL_UNDEF(f_name);
3345:         ZVAL_UNDEF(f_name);
3346:         ZVAL_UNDEF(f_name);
3347:         ZVAL_UNDEF(f_name);
3348:         ZVAL_UNDEF(f_name);
3349:         ZVAL_UNDEF(f_name);
3350:         ZVAL_UNDEF(f_name);
3351:         ZVAL_UNDEF(f_name);
3352:         ZVAL_UNDEF(f_name);
3353:         ZVAL_UNDEF(f_name);
3354:         ZVAL_UNDEF(f_name);
3355:         ZVAL_UNDEF(f_name);
3356:         ZVAL_UNDEF(f_name);
3357:         ZVAL_UNDEF(f_name);
3358:         ZVAL_UNDEF(f_name);
3359:         ZVAL_UNDEF(f_name);
3360:         ZVAL_UNDEF(f_name);
3361:         ZVAL_UNDEF(f_name);
3362:         ZVAL_UNDEF(f_name);
3363:         ZVAL_UNDEF(f_name);
3364:         ZVAL_UNDEF(f_name);
3365:         ZVAL_UNDEF(f_name);
3366:         ZVAL_UNDEF(f_name);
3367:         ZVAL_UNDEF(f_name);
3368:         ZVAL_UNDEF(f_name);
3369:         ZVAL_UNDEF(f_name);
3370:         ZVAL_UNDEF(f_name);
3371:         ZVAL_UNDEF(f_name);
3372:         ZVAL_UNDEF(f_name);
3373:         ZVAL_UNDEF(f_name);
3374:         ZVAL_UNDEF(f_name);
3375:         ZVAL_UNDEF(f_name);
3376:         ZVAL_UNDEF(f_name);
3377:         ZVAL_UNDEF(f_name);
3378:         ZVAL_UNDEF(f_name);
3379:         ZVAL_UNDEF(f_name);
3380:         ZVAL_UNDEF(f_name);
3381:         ZVAL_UNDEF(f_name);
3382:         ZVAL_UNDEF(f_name);
3383:         ZVAL_UNDEF(f_name);
3384:         ZVAL_UNDEF(f_name);
3385:         ZVAL_UNDEF(f_name);
3386:         ZVAL_UNDEF(f_name);
3387:         ZVAL_UNDEF(f_name);
3388:         ZVAL_UNDEF(f_name);
3389:         ZVAL_UNDEF(f_name);
3390:         ZVAL_UNDEF(f_name);
3391:         ZVAL_UNDEF(f_name);
3392:         ZVAL_UNDEF(f_name);
3393:         ZVAL_UNDEF(f_name);
3394:         ZVAL_UNDEF(f_name);
3395:         ZVAL_UNDEF(f_name);
3396:         ZVAL_UNDEF(f_name);
3397:         ZVAL_UNDEF(f_name);
3398:         ZVAL_UNDEF(f_name);
3399:         ZVAL_UNDEF(f_name);
3400:         ZVAL_UNDEF(f_name);
3401:         ZVAL_UNDEF(f_name);
3402:         ZVAL_UNDEF(f_name);
3403:         ZVAL_UNDEF(f_name);
3404:         ZVAL_UNDEF(f_name);
3405:         ZVAL_UNDEF(f_name);
3406:         ZVAL_UNDEF(f_name);
3407:         ZVAL_UNDEF(f_name);
3408:         ZVAL_UNDEF(f_name);
3409:         ZVAL_UNDEF(f_name);
3410:         ZVAL_UNDEF(f_name);
3411:         ZVAL_UNDEF(f_name);
3412:         ZVAL_UNDEF(f_name);
3413:         ZVAL_UNDEF(f_name);
3414:         ZVAL_UNDEF(f_name);
3415:         ZVAL_UNDEF(f_name);
3416:         ZVAL_UNDEF(f_name);
3417:         ZVAL_UNDEF(f_name);
3418:         ZVAL_UNDEF(f_name);
3419:         ZVAL_UNDEF(f_name);
3420:         ZVAL_UNDEF(f_name);
3421:         ZVAL_UNDEF(f_name);
3422:         ZVAL_UNDEF(f_name);
3423:         ZVAL_UNDEF(f_name);
3424:         ZVAL_UNDEF(f_name);
3425:         ZVAL_UNDEF(f_name);
3426:         ZVAL_UNDEF(f_name);
3427:         ZVAL_UNDEF(f_name);
3428:         ZVAL_UNDEF(f_name);
3429:         ZVAL_UNDEF(f_name);
3430:         ZVAL
```

```

1500: struct {
1501:     HashTable *new_ht = 2_ARRAYVAL_P(2_REFVAL(params[0]));
1502:     if (aht != new_ht) {
1503:         spl_array_wipeout_hash_table(intern, new_ht);
1504:     } else {
1505:         GC_DEREF(aht);
1506:     }
1507:     cfree(2_REF(params[0]));
1508:     zend_string_free(2_STR(function_name));
1509: }
1510: /* }}} */
1511: #define SPL_ARRAY_METHOD(name, fname, usa_arg) \
1512:     static inline void INTERNAL_FUNCTION_PARAM_PASSTHRU, (name, ainfo, &fname)-1, usa_arg); \
1513: }
1514:
1515: /* {{{ proto int ArrayObject::asort([int $sort_flags = SORT_REGULAR])
1516:  * proto int ArrayIterator::asort([int $sort_flags = SORT_REGULAR])
1517:  * Sort the entries by values. */
1518: SPL_ARRAY_METHOD(Array, asort, SPL_ARRAY_METHOD_MAY_USER_ARG) /* }}} */
1519:
1520: /* {{{ proto int ArrayObject::ksort([int $sort_flags = SORT_REGULAR])
1521:  * proto int ArrayIterator::ksort([int $sort_flags = SORT_REGULAR])
1522:  * Sort the entries by key. */
1523: SPL_ARRAY_METHOD(Array, ksort, SPL_ARRAY_METHOD_MAY_USER_ARG) /* }}} */
1524:
1525: /* {{{ proto int ArrayObject::uasort(callback cmp_function)
1526:  * proto int ArrayIterator::uasort(callback cmp_function)
1527:  * Sort the entries by values user defined function. */
1528: SPL_ARRAY_METHOD(Array, uasort, SPL_ARRAY_METHOD_USER_ARG) /* }}} */
1529:
1530: /* {{{ proto int ArrayObject::uksort(callback cmp_function)
1531:  * proto int ArrayIterator::uksort(callback cmp_function)
1532:  * Sort the entries by key using user defined function. */
1533: SPL_ARRAY_METHOD(Array, uksort, SPL_ARRAY_METHOD_USER_ARG) /* }}} */
1534:
1535: /* {{{ proto int ArrayObject::natasort()
1536:  * proto int ArrayIterator::natasort()
1537:  * Sort the entries by values using "natural order" algorithm. */
1538: SPL_ARRAY_METHOD(Array, natasort, SPL_ARRAY_METHOD_NO_ARG) /* }}} */
1539:
1540: /* {{{ proto int ArrayObject::natasort()
1541:  * proto int ArrayIterator::natasort()
1542:  * Sort the entries by key using case insensitive "natural order" algorithm. */
1543: SPL_ARRAY_METHOD(Array, natasort, SPL_ARRAY_METHOD_NO_ARG) /* }}} */
1544:
1545: /* {{{ proto mixed NULL ArrayIterator::current()
1546:  * Return current array entry */
1547: SPL_METHOD(Array, current)
1548: {
1549:     zval *object = getThis();
1550:     spl_array_object *intern = 2_SPLARRAY_P(object);
1551:     zval *entry;
1552:     HashTable *aht = spl_array_get_hash_table(intern);
1553:     if (zend_parse_parameters_none() == FAILURE) {
1554:         return;
1555:     }
1556:     if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1557:         return;
1558:     }
1559:     if (entry = zend_hash_get_current_data_ex(aht, spl_array_get_pos_ptr(aht, intern)) == NULL) {
1560:         return;
1561:     }
1562:     if (Z_TYPE_P(entry) == IS_INDIRECT) {
1563:         entry = Z_INDIRECT_P(entry);
1564:     }
1565:     if (Z_TYPE_P(entry) == IS_UNDEF) {
1566:         return;
1567:     }
1568:     ZVAL_DEREF(entry);
1569:     ZVAL_COPY(return_value, entry);
1570: }
1571: /* }}} */
1572:
1573: /* {{{ proto mixed NULL ArrayIterator::key()
1574:  * Return current array key */
1575: SPL_METHOD(Array, key)
1576: {
1577:     if (zend_parse_parameters_none() == FAILURE) {
1578:         return;
1579:     }
1580:     spl_array_iterator_key(getThis(), return_value);
1581:     /* }}} */
1582:
1583: void spl_array_iterator_key(zval *object, zval *return_value) /* {{{ */
1584: {
1585:     spl_array_object *intern = 2_SPLARRAY_P(object);
1586:     HashTable *aht = spl_array_get_hash_table(intern);
1587:     if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1588:         return;
1589:     }
1590:     zend_hash_get_current_key_zval_ex(aht, return_value, spl_array_get_pos_ptr(aht, intern));
1591:     /* }}} */
1592:
1593: /* {{{ proto void ArrayIterator::next()
1594:  * Move to next entry */
1595: SPL_METHOD(Array, next)
1596: {
1597:     zval *object = getThis();
1598:     spl_array_object *intern = 2_SPLARRAY_P(object);
1599:     HashTable *aht = spl_array_get_hash_table(intern);
1600:     if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1601:         return;
1602:     }
1603:     zend_hash_get_current_key_zval_ex(aht, return_value, spl_array_get_pos_ptr(aht, intern));
1604:     /* }}} */
1605:
1606: /* {{{ proto bool ArrayIterator::valid()
1607:  * Check whether array contains more entries */
1608: SPL_METHOD(Array, valid)
1609: {
1610:     zval *object = getThis();
1611:     spl_array_object *intern = 2_SPLARRAY_P(object);
1612:     HashTable *aht = spl_array_get_hash_table(intern);
1613:     if (zend_parse_parameters_none() == FAILURE) {
1614:         return;
1615:     }
1616:     if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1617:         return;
1618:     }
1619:     spl_array_next_ex(intern, aht);
1620:     /* }}} */
1621:
1622: /* {{{ proto bool ArrayIterator::valid()
1623:  * Check whether array contains more entries */
1624: SPL_METHOD(Array, valid)
1625: {
1626:     zval *object = getThis();
1627:     spl_array_object *intern = 2_SPLARRAY_P(object);
1628:     HashTable *aht = spl_array_get_hash_table(intern);
1629:     if (zend_parse_parameters_none() == FAILURE) {
1630:         return;
1631:     }
1632:     if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1633:         return;
1634:     }
1635:     RETURN_FALSE;
1636: } else {
1637:     RETURN_BOOL(zend_hash_has_more_elements_ex(aht, spl_array_get_pos_ptr(aht, intern)) == SUCCESS);
1638: }
1639: /* }}} */
1640:
1641: /* {{{ proto bool RecursiveArrayIterator::hasChildren()
1642:  * Check whether current element has children (e.g. is an array) */
1643: SPL_METHOD(Array, hasChildren)
1644: {
1645:     zval *object = getThis(), *entry;
1646:     spl_array_object *intern = 2_SPLARRAY_P(object);
1647:     HashTable *aht = spl_array_get_hash_table(intern);
1648:     if (zend_parse_parameters_none() == FAILURE) {
1649:         return;
1650:     }
1651:     if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1652:         return;
1653:     }
1654:     if (entry = zend_hash_get_current_data_ex(aht, spl_array_get_pos_ptr(aht, intern)) == NULL) {
1655:         return;
1656:     }
1657:     if (Z_TYPE_P(entry) == IS_INDIRECT) {
1658:         entry = Z_INDIRECT_P(entry);
1659:     }
1660:     ZVAL_DEREF(entry);
1661:     RETURN_BOOL(Z_TYPE_P(entry) == IS_ARRAY || (Z_TYPE_P(entry) == IS_OBJECT && (intern->var_flags & SPL_ARRAY_CHILD_ARRAYS_ONLY) == 0));
1662: }
1663: /* }}} */
1664:
1665: /* {{{ proto object RecursiveArrayIterator::getChildren()
1666:  * Create a new iterator for the current element (same class as $this) */
1667: SPL_METHOD(Array, getChildren)
1668: {
1669:     zval *object = getThis(), *entry, flags;
1670:     spl_array_object *intern = 2_SPLARRAY_P(object);
1671:     HashTable *aht = spl_array_get_hash_table(intern);
1672:     if (zend_parse_parameters_none() == FAILURE) {
1673:         return;
1674:     }
1675:     if (spl_array_object_verify_pos(intern, aht) == FAILURE) {
1676:         return;
1677:     }
1678: }

```

```

16891:
16892:
16893:     if (entry == send_ssh_get_current_data_axiath, spi_array_get_pos_ptr(ht, intern)) == NULL) {
16894:         return;
16895:     }
16896:
16897:     if (IS_TYPE_P(entry) == IS_INDIRECT) {
16898:         entry = IS_INDIRECT_P(entry);
16899:     }
16900:
16901:     EVAL_UNDEF(entry);
16902:
16903:     if (IS_TYPE_P(entry) == IS_OBJECT) {
16904:         if ((intern->var_flags & SPL_ARRAY_CHILD_ARRAYS_ONLY) != 0) {
16905:             return;
16906:         }
16907:
16908:         if (instanceof_function(S_OBJECT_P(entry), S_OBJECT_P(getthis()))) {
16909:             EVAL_COPY(return_value, S_OBJ_P(entry));
16910:             S_ADOBEF_P(return_value);
16911:             return;
16912:         }
16913:     }
16914:
16915:     EVAL_LONG(flags, intern->var_flags);
16916:     spl_instantiate_arg_ax2(S_OBJECT_P(getthis()), return_value, entry, flags);
16917: }
16918:
16919: /* }}} */
16920:
16921: /* {{{ proto string ArrayObject::serialize()
16922:  * Serialize the object */
16923:
16924: SPL_METHOD(Array, serialize)
16925: {
16926:     zval *object = getThis();
16927:     spl_array_object *intern = S_SPLARRAY_P(object);
16928:     HashTable *ht = spl_array_get_hash_table(intern);
16929:     zend_members *flags;
16930:     php_serialize_data_t var_hash;
16931:     smart_str buf = {0};
16932:
16933:     if (zend_parse_parameters_none() == FAILURE) {
16934:         return;
16935:     }
16936:
16937:     if (ht) {
16938:         zend_error_core(NULL, E_NOTICE, "Array was modified outside object and is no longer an array");
16939:         return;
16940:     }
16941:
16942:     PHP_VAR_SERIALIZE_INIT(var_hash);
16943:
16944:     EVAL_LONG(flags, (intern->var_flags & SPL_ARRAY_CLONE_MASK));
16945:
16946:     /* storage */
16947:     smart_str_append(&buf, "a:", 2);
16948:     php_var_serialize(&buf, &flags, var_hash);
16949:
16950:     if ((intern->var_flags & SPL_ARRAY_IS_SELF) &
16951:         php_var_serialize(&buf, &intern->array, &var_hash)) {
16952:         smart_str_append(&buf, ' ');
16953:     }
16954:
16955:     /* members */
16956:     smart_str_append(&buf, "m:", 2);
16957:     if ((intern->std.properties) &
16958:         rebuild_obj_std_properties(&intern->std)) {
16959:     }
16960:
16961:     EVAL_ARR(&members, intern->std.properties);
16962:     php_var_serialize(&buf, &members, &var_hash); /* finishes the string */
16963:
16964:     /* done */
16965:     if (buf.s) {
16966:         RETURN_NEW_STR(&buf.s);
16967:     }
16968:     RETURN_NULL();
16969: }
16970:
16971: /* }}} */
16972:
16973: /* {{{ proto void ArrayObject::unserialize(string serialized)
16974:  * unserialize the object */
16975:
16976: SPL_METHOD(Array, unserialize)
16977: {
16978:     zval *object = getThis();
16979:     spl_array_object *intern = S_SPLARRAY_P(object);
16980:
16981:     char *buf;
16982:     const buf_len;
16983:     const unsigned char *p;
16984:     php_unserialize_data_t var_hash;
16985:     zend_members *flags;
16986:     zend_long flags;
16987:
16988:     if (zend_parse_parameters(ZEND_NUM_ARGS() & "s", &buf, &buf_len) == FAILURE) {
16989:         return;
16990:     }
16991:
16992:     if (buf_len == 0) {
16993:         return;
16994:     }
16995:
16996:     if (intern->copyCount > 0) {
16997:         zend_error(E_WARNING, "Modification of ArrayObject during sorting is prohibited");
16998:         return;
16999:     }
17000:
17001:     /* storage */
17002:     s = p = (const unsigned char *)buf;
17003:     PHP_VAR_UNSERIALIZE_INIT(var_hash);
17004:
17005:     if (*p != 'a' || **p != ':') {
17006:         goto outexcept;
17007:     }
17008:     *p++;
17009:
17010:     &flags = var_tmp_var(&var_hash);
17011:     if (php_var_unserialize(&flags, &p, s + buf_len, &var_hash) || IS_TYPE_P(&flags) != IS_LONG) {
17012:     }
17013:
17014:     /*
17015:      * "p" is for "p"
17016:      * flags = S_ARRAY_P(&flags);
17017:      * flags needs to be verified and we also need to verify whether the next
17018:      * thing we get is "p". After that we require an "a" or something else
17019:      * where "a" stands for members and anything else should be an array. If
17020:      * neither "a" or "a" follows we have an error. */
17021:
17022:     if (*p != ':') {
17023:         goto outexcept;
17024:     }
17025:
17026:     *p++;
17027:
17028:     if (flags & SPL_ARRAY_IS_SELF) {
17029:         /* If IS_SELF is used, the flags are not followed by an array/object */
17030:         &intern->var_flags = "SPL_ARRAY_CLONE_MASK";
17031:         &intern->var_flags |= flags & SPL_ARRAY_CLONE_MASK;
17032:         &val_get_std(&intern->array) = flags & SPL_ARRAY_CLONE_MASK;
17033:         EVAL_UNDEF(&intern->array);
17034:     } else {
17035:         if (*p != 'a' || **p != 'O' || *p != 'O' || *p != 'C') {
17036:             goto outexcept;
17037:         }
17038:
17039:         array = var_tmp_var(&var_hash);
17040:         if (php_var_unserialize(&array, &p, s + buf_len, &var_hash)) {
17041:             if (IS_TYPE_P(array) != IS_ARRAY & IS_TYPE_P(array) != IS_OBJECT) {
17042:                 goto outexcept;
17043:             }
17044:
17045:             &intern->var_flags = "SPL_ARRAY_CLONE_MASK";
17046:             &intern->var_flags |= flags & SPL_ARRAY_CLONE_MASK;
17047:
17048:             if (IS_TYPE_P(array) == IS_ARRAY) {
17049:                 &val_get_std(&intern->array);
17050:                 EVAL_COPY(&intern->array, array);
17051:             } else {
17052:                 spl_array_set_array(object, &intern, array, CL_1);
17053:             }
17054:
17055:             if (*p != ':') {
17056:                 goto outexcept;
17057:             }
17058:             *p++;
17059:
17060:             /* members */
17061:             if (*p != 'a' || **p != ':') {
17062:                 goto outexcept;
17063:             }
17064:             *p++;
17065:
17066:             &members = var_tmp_var(&var_hash);
17067:             if (php_var_unserialize(&members, &p, s + buf_len, &var_hash) || IS_TYPE_P(&members) != IS_ARRAY) {
17068:                 goto outexcept;
17069:             }
17070:
17071:             /* copy members */
17072:             object_properties_load(&intern->std, &INTERNAL_P(&members));
17073:
17074:             /* done reading Serialized */
17075:             PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
17076:
17077:             outexcept;
17078:
17079:             PHP_VAR_UNSERIALIZE_DESTROY(var_hash);
17080:             zend_throw_exception(spl_ce UnexpectedValueException, 0, "Error at offset " ZEND_LONG_FMT

```

```
1881:     return;
1882:
1883: } /* }}} */
1884:
1885: /* {{{ arginfo and function table */
1886: ZEND_BEGIN_ARG_INFO_EX(arginfo_array___construct, 0, 0, 0)
1887:     ZEND_ARG_INFO(0, array)
1888:     ZEND_ARG_INFO(0, ar_flags)
1889:     ZEND_ARG_INFO(0, iterator_class)
1890: ZEND_END_ARG_INFO()
1891:
1892: /* ArrayIterator::__construct and ArrayObject::__construct have different signatures */
1893: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_iterator___construct, 0, 0, 0)
1894:     ZEND_ARG_INFO(0, array)
1895:     ZEND_ARG_INFO(0, ar_flags)
1896: ZEND_END_ARG_INFO()
1897:
1898: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_offsetGet, 0, 0, 1)
1899:     ZEND_ARG_INFO(0, index)
1900: ZEND_END_ARG_INFO()
1901:
1902: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_offsetSet, 0, 0, 2)
1903:     ZEND_ARG_INFO(0, index)
1904:     ZEND_ARG_INFO(0, newval)
1905: ZEND_END_ARG_INFO()
1906:
1907: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_append, 0)
1908:     ZEND_ARG_INFO(0, value)
1909: ZEND_END_ARG_INFO()
1910:
1911: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_seek, 0)
1912:     ZEND_ARG_INFO(0, position)
1913: ZEND_END_ARG_INFO()
1914:
1915: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_exchangeArray, 0)
1916:     ZEND_ARG_INFO(0, array)
1917: ZEND_END_ARG_INFO()
1918:
1919: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_setFlags, 0)
1920:     ZEND_ARG_INFO(0, flags)
1921: ZEND_END_ARG_INFO()
1922:
1923: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_getIteratorClass, 0)
1924:     ZEND_ARG_INFO(0, iteratorClass)
1925: ZEND_END_ARG_INFO()
1926:
1927: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_uksort, 0)
1928:     ZEND_ARG_INFO(0, cmp_function)
1929: ZEND_END_ARG_INFO()
1930:
1931: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_unserialize, 0)
1932:     ZEND_ARG_INFO(0, serialized)
1933: ZEND_END_ARG_INFO()
1934:
1935: ZEND_BEGIN_ARG_INFO_EX(arginfo_array_void, 0)
1936: ZEND_END_ARG_INFO()
1937:
1938: static const zend_function_entry spl_funcs_arrayObject[] = {
1939:     SPL_ME(Array, __construct, arginfo_array___construct, ZEND_ACC_PUBLIC)
1940:     SPL_ME(Array, offsetExists, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1941:     SPL_ME(Array, offsetGet, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1942:     SPL_ME(Array, offsetSet, arginfo_array_offsetSet, ZEND_ACC_PUBLIC)
1943:     SPL_ME(Array, offsetUnset, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1944:     SPL_ME(Array, append, arginfo_array_append, ZEND_ACC_PUBLIC)
1945:     SPL_ME(Array, getArrayCopy, arginfo_array_void, ZEND_ACC_PUBLIC)
1946:     SPL_ME(Array, count, arginfo_array_void, ZEND_ACC_PUBLIC)
1947:     SPL_ME(Array, getFlags, arginfo_array_void, ZEND_ACC_PUBLIC)
1948:     SPL_ME(Array, setFlags, arginfo_array_setFlags, ZEND_ACC_PUBLIC)
1949:     SPL_ME(Array, asort, arginfo_array_void, ZEND_ACC_PUBLIC)
1950:     SPL_ME(Array, usort, arginfo_array_void, ZEND_ACC_PUBLIC)
1951:     SPL_ME(Array, uasort, arginfo_array_uksort, ZEND_ACC_PUBLIC)
1952:     SPL_ME(Array, uksort, arginfo_array_uksort, ZEND_ACC_PUBLIC)
1953:     SPL_ME(Array, natsort, arginfo_array_void, ZEND_ACC_PUBLIC)
1954:     SPL_ME(Array, natcasesort, arginfo_array_void, ZEND_ACC_PUBLIC)
1955:     SPL_ME(Array, unserialize, arginfo_array_unserialize, ZEND_ACC_PUBLIC)
1956:     SPL_ME(Array, serialize, arginfo_array_void, ZEND_ACC_PUBLIC)
1957:     /* ArrayObject specific */
1958:     SPL_ME(Array, getIterator, arginfo_array_void, ZEND_ACC_PUBLIC)
1959:     SPL_ME(Array, exchangeArray, arginfo_array_exchangeArray, ZEND_ACC_PUBLIC)
1960:     SPL_ME(Array, setIteratorClass, arginfo_array_getIteratorClass, ZEND_ACC_PUBLIC)
1961:     SPL_ME(Array, getIteratorClass, arginfo_array_void, ZEND_ACC_PUBLIC)
1962:     PHP_FE_END
1963: };
1964:
1965: static const zend_function_entry spl_funcs_arrayIterator[] = {
1966:     SPL_ME(ArrayIterator, __construct, arginfo_array_iterator___construct, ZEND_ACC_PUBLIC)
1967:     SPL_ME(Array, offsetExists, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1968:     SPL_ME(Array, offsetGet, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1969:     SPL_ME(Array, offsetSet, arginfo_array_offsetSet, ZEND_ACC_PUBLIC)
1970:     SPL_ME(Array, offsetUnset, arginfo_array_offsetGet, ZEND_ACC_PUBLIC)
1971:     SPL_ME(Array, append, arginfo_array_append, ZEND_ACC_PUBLIC)
1972:     SPL_ME(Array, getArrayCopy, arginfo_array_void, ZEND_ACC_PUBLIC)
1973:     SPL_ME(Array, count, arginfo_array_void, ZEND_ACC_PUBLIC)
1974:     SPL_ME(Array, getFlags, arginfo_array_void, ZEND_ACC_PUBLIC)
1975:     SPL_ME(Array, setFlags, arginfo_array_setFlags, ZEND_ACC_PUBLIC)
1976:     SPL_ME(Array, asort, arginfo_array_void, ZEND_ACC_PUBLIC)
1977:     SPL_ME(Array, usort, arginfo_array_void, ZEND_ACC_PUBLIC)
1978:     SPL_ME(Array, uasort, arginfo_array_uksort, ZEND_ACC_PUBLIC)
1979:     SPL_ME(Array, uksort, arginfo_array_uksort, ZEND_ACC_PUBLIC)
1980:     SPL_ME(Array, natsort, arginfo_array_void, ZEND_ACC_PUBLIC)
1981:     SPL_ME(Array, natcasesort, arginfo_array_void, ZEND_ACC_PUBLIC)
1982:     SPL_ME(Array, unserialize, arginfo_array_unserialize, ZEND_ACC_PUBLIC)
1983:     SPL_ME(Array, serialize, arginfo_array_void, ZEND_ACC_PUBLIC)
1984:     /* ArrayIterator specific */
1985:     SPL_ME(Array, rewind, arginfo_array_void, ZEND_ACC_PUBLIC)
1986:     SPL_ME(Array, current, arginfo_array_void, ZEND_ACC_PUBLIC)
1987:     SPL_ME(Array, key, arginfo_array_void, ZEND_ACC_PUBLIC)
1988:     SPL_ME(Array, next, arginfo_array_void, ZEND_ACC_PUBLIC)
1989:     SPL_ME(Array, valid, arginfo_array_void, ZEND_ACC_PUBLIC)
1990:     SPL_ME(Array, seek, arginfo_array_seek, ZEND_ACC_PUBLIC)
1991:     PHP_FE_END
1992: };
1993:
1994: static const zend_function_entry spl_funcs_recursiveArrayIterator[] = {
1995:     SPL_ME(Array, hasChildren, arginfo_array_void, ZEND_ACC_PUBLIC)
1996:     SPL_ME(Array, getChildren, arginfo_array_void, ZEND_ACC_PUBLIC)
1997:     PHP_FE_END
1998: };
1999: /* }}} */
2000:
2001: /* {{{ PHP_MINIT_FUNCTION(spl_array) */
2002: PHP_MINIT_FUNCTION(spl_array)
2003: {
2004:     REGISTER_SPL_STD_CLASS_EX(ArrayObject, spl_array_object_new, spl_funcs_arrayObject);
2005:     REGISTER_SPL_IMPLEMENTATIONS(ArrayObject, Aggregate);
2006:     REGISTER_SPL_IMPLEMENTATIONS(ArrayObject, ArrayAccess);
2007:     REGISTER_SPL_IMPLEMENTATIONS(ArrayObject, Serializable);
2008:     REGISTER_SPL_IMPLEMENTATIONS(ArrayObject, Countable);
2009:     memcpy(spl_handler_arrayObject, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
2010:
2011:     spl_handler_arrayObject->offset = XTOffsetOf(spl_array_object, std);
2012:
2013:     spl_handler_arrayObject->clone_obj = spl_array_object_clone;
2014:     spl_handler_arrayObject->read_dimension = spl_array_read_dimension;
2015:     spl_handler_arrayObject->write_dimension = spl_array_write_dimension;
2016:     spl_handler_arrayObject->unset_dimension = spl_array_unset_dimension;
2017:     spl_handler_arrayObject->has_dimension = spl_array_has_dimension;
2018:     spl_handler_arrayObject->count_elements = spl_array_object_count_elements;
2019:
2020:     spl_handler_arrayObject->get_properties = spl_array_get_properties;
2021:     spl_handler_arrayObject->get_debug_info = spl_array_get_debug_info;
2022:     spl_handler_arrayObject->get_gc = spl_array_get_gc;
2023:     spl_handler_arrayObject->read_property = spl_array_read_property;
2024:     spl_handler_arrayObject->write_property = spl_array_write_property;
2025:     spl_handler_arrayObject->get_property_ptr_ptr = spl_array_get_property_ptr_ptr;
2026:     spl_handler_arrayObject->has_property = spl_array_has_property;
2027:     spl_handler_arrayObject->unset_property = spl_array_unset_property;
2028:
2029:     spl_handler_arrayObject->compare_objects = spl_array_compare_objects;
2030:     spl_handler_arrayObject->dtor_obj = zend_objects_destroy_obj;
2031:     spl_handler_arrayObject->free_obj = spl_array_object_free_storage;
2032:
2033:     REGISTER_SPL_STD_CLASS_EX(ArrayIterator, spl_array_iterator_new, spl_funcs_arrayIterator);
2034:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, Iterator);
2035:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, ArrayAccess);
2036:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, SeekableIterator);
2037:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, Serializable);
2038:     REGISTER_SPL_IMPLEMENTATIONS(ArrayIterator, Countable);
2039:     memcpy(spl_handler_arrayIterator, spl_handler_arrayObject, sizeof(zend_object_handlers));
2040:     spl_array_iterator->get_iterator = spl_array_get_iterator;
2041:
2042:     REGISTER_SPL_CLASS_CONST_LONG(ArrayObject, "STD_PROP_LIST", SPL_ARRAY_STD_PROP_LIST);
2043:     REGISTER_SPL_CLASS_CONST_LONG(ArrayObject, "ARRAY_AS_PROPS", SPL_ARRAY_ARRAY_AS_PROPS);
2044:
2045:     REGISTER_SPL_CLASS_CONST_LONG(ArrayIterator, "STD_PROP_LIST", SPL_ARRAY_STD_PROP_LIST);
2046:     REGISTER_SPL_CLASS_CONST_LONG(ArrayIterator, "ARRAY_AS_PROPS", SPL_ARRAY_ARRAY_AS_PROPS);
2047:
2048:     REGISTER_SPL_SUB_CLASS_EX(RecursiveArrayIterator, ArrayIterator, spl_array_iterator_new, spl_funcs_recursiveArrayIterator);
2049:     REGISTER_SPL_IMPLEMENTATIONS(RecursiveArrayIterator, RecursiveIterator);
2050:     spl_recursiveArrayIterator->get_iterator = spl_array_get_iterator;
2051:
2052:     REGISTER_SPL_CLASS_CONST_LONG(RecursiveArrayIterator, "CHILD_ARRAYS_ONLY", SPL_ARRAY_CHILD_ARRAYS_ONLY);
2053:
2054:     return SUCCESS;
2055: }
2056: /* }}} */
2057:
2058: /*
2059:  * Local Variables:
2060:  * tab-width: 4
2061:  * c-basic-offset: 4
2062:  * End:
2063:  * vim600: fdm=marker
2064:  * vim: noet sw=4 ts=4
2065:  */

```

```
1: /*
2:  *-----*
3:  * | PHP Version 7 |
4:  *-----*
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  *-----*
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  *-----*
15:  * | Authors: Etienne Kneuss <colder@php.net> |
16:  *-----*
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_HEAP_H
22: #define SPL_HEAP_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_SplHeap;
28: extern PHPAPI zend_class_entry *spl_ce_SplMinHeap;
29: extern PHPAPI zend_class_entry *spl_ce_SplMaxHeap;
30:
31: extern PHPAPI zend_class_entry *spl_ce_SplPriorityQueue;
32:
33: PHP_MINIT_FUNCTION(spl_heap);
34:
35: #endif /* SPL_HEAP_H */
36:
37: /*
38:  * Local Variables:
39:  * c-basic-offset: 4
40:  * tab-width: 4
41:  * End:
42:  * vim600: fdm=marker
43:  * vim: noet sw=4 ts=4
44:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_ENGINE_H
22: #define SPL_ENGINE_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26: #include "zend_interfaces.h"
27:
28: PHPAPI void spl_instantiate(zend_class_entry *pce, zval *object);
29:
30: PHPAPI zend_long spl_offset_convert_to_long(zval *offset);
31:
32: /* {{{ spl_instantiate_arg_ex1 */
33: static inline int spl_instantiate_arg_ex1(zend_class_entry *pce, zval *retval, zval *arg1)
34: {
35:     zend_function *func = pce->constructor;
36:     spl_instantiate(pce, retval);
37:
38:     zend_call_method(retval, pce, &func, ZEND_VAL(func->common.function_name), ZEND_LEN(func->common.function_name), NULL, 1, arg1, NULL);
39:     return 0;
40: }
41: /* }}} */
42:
43: /* {{{ spl_instantiate_arg_ex2 */
44: static inline int spl_instantiate_arg_ex2(zend_class_entry *pce, zval *retval, zval *arg1, zval *arg2)
45: {
46:     zend_function *func = pce->constructor;
47:     spl_instantiate(pce, retval);
48:
49:     zend_call_method(retval, pce, &func, ZEND_VAL(func->common.function_name), ZEND_LEN(func->common.function_name), NULL, 2, arg1, arg2);
50:     return 0;
51: }
52: /* }}} */
53:
54: /* {{{ spl_instantiate_arg_n */
55: static inline void spl_instantiate_arg_n(zend_class_entry *pce, zval *retval, int argc, zval *argv)
56: {
57:     zend_function *func = pce->constructor;
58:     zend_fcall_info fci;
59:     zend_fcall_info_cache fcc;
60:     zval dummy;
61:
62:     spl_instantiate(pce, retval);
63:
64:     fci.size = sizeof(zend_fcall_info);
65:     ZVAL_STR(&fci.function_name, func->common.function_name);
66:     fci.object = Z_OBJ_P(retval);
67:     fci.retval = &dummy;
68:     fci.param_count = argc;
69:     fci.params = argv;
70:     fci.no_separation = 1;
71:
72:     fcc.function_handler = func;
73:     fcc.calling_scope = zend_get_executed_scope();
74:     fcc.called_scope = pce;
75:     fcc.object = Z_OBJ_P(retval);
76:
77:     zend_call_function(&fci, &fcc);
78: }
79: /* }}} */
80:
81: #endif /* SPL_ENGINE_H */
82:
83: /*
84:  * Local Variables:
85:  * n-basis-offset: 4
86:  * tab-width: 4
87:  * End:
88:  * vim600: fdm=marker
89:  * vim: noet sw=4 ts=4
90:  */
```



```

1: /*
2:  *
3:  * PHP Version 7
4:  *
5:  * Copyright (c) 1997-2018 The PHP Group
6:  *
7:  * This source file is subject to version 3.01 of the PHP license,
8:  * that is bundled with this package in the file LICENSE, and is
9:  * available through the world-wide-web at the following url:
10:  * http://www.php.net/license/3.01.txt
11:  * If you did not receive a copy of the PHP license and are unable to
12:  * obtain it through the world-wide-web, please send a note to
13:  * license@php.net so we can mail you a copy immediately.
14:  *
15:  * Authors: Marcus Boerger <helly@php.net>
16:  */
17:
18:
19: /* $Id$ */
20:
21: #ifndef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "php.h"
26: #include "php_ini.h"
27: #include "ext/standard/info.h"
28: #include "zend_exceptions.h"
29: #include "zend_interfaces.h"
30:
31: #include "spl_api.h"
32: #include "spl_functions.h"
33: #include "spl_engine.h"
34: #include "spl_iterators.h"
35: #include "spl_directory.h"
36: #include "spl_array.h"
37: #include "spl_exceptions.h"
38: #include "zend_smart_str.h"
39:
40: #ifndef HAVE_ACCOPT
41: #define HAVE_ACCOPT
42: #endif
43:
44: #define PHPAPI zend_class_entry *spl_ce_RecursiveIterator;
45: #define PHPAPI zend_class_entry *spl_ce_RecursiveIteratorIterator;
46: #define PHPAPI zend_class_entry *spl_ce_RecursiveIterator;
47: #define PHPAPI zend_class_entry *spl_ce_CallbackFilterIterator;
48: #define PHPAPI zend_class_entry *spl_ce_RecursiveFilterIterator;
49: #define PHPAPI zend_class_entry *spl_ce_RecursiveCallbackFilterIterator;
50: #define PHPAPI zend_class_entry *spl_ce_ParentIterator;
51: #define PHPAPI zend_class_entry *spl_ce_SeekableIterator;
52: #define PHPAPI zend_class_entry *spl_ce_OuterIterator;
53: #define PHPAPI zend_class_entry *spl_ce_CachingIterator;
54: #define PHPAPI zend_class_entry *spl_ce_RecursiveCachingIterator;
55: #define PHPAPI zend_class_entry *spl_ce_OuterIterator;
56: #define PHPAPI zend_class_entry *spl_ce_IteratorIterator;
57: #define PHPAPI zend_class_entry *spl_ce_NoRewindIterator;
58: #define PHPAPI zend_class_entry *spl_ce_InfiniteIterator;
59: #define PHPAPI zend_class_entry *spl_ce_EmptyIterator;
60: #define PHPAPI zend_class_entry *spl_ce_AppendIterator;
61: #define PHPAPI zend_class_entry *spl_ce_DequeIterator;
62: #define PHPAPI zend_class_entry *spl_ce_RecursiveDequeIterator;
63: #define PHPAPI zend_class_entry *spl_ce_RecursiveTraversable;
64:
65: #define ZEND_BEGIN_ARG_INFO_EX(arginfo_recursive_it_void, 0)
66: #define ZEND_ARG_END_INFO()
67:
68: static const zend_function_entry spl_func RecursiveIterator[] = {
69:     SPL_ABSTRACT_METHOD(RecursiveIterator, hasChildren, arginfo_recursive_it_void)
70:     SPL_ABSTRACT_METHOD(RecursiveIterator, getChildren, arginfo_recursive_it_void)
71:     PHP_FE_END
72: };
73:
74: typedef enum {
75:     RIT_LEAVES_ONLY = 0,
76:     RIT_SELF_FIRST = 1,
77:     RIT_CHILD_FIRST = 2
78: } RecursiveIteratorMode;
79:
80: #define RIT_CATCH_GET_CHILD CATCH_GET_CHILD
81:
82: typedef enum {
83:     RIT_ITPASS_CURRENT = 4,
84:     RIT_ITPASS_KEY = 8
85: } RecursiveTraversableFlags;
86:
87: typedef enum {
88:     RS_NEXT = 0,
89:     RS_TEST = 1,
90:     RS_SELF = 2,
91:     RS_CHILD = 3,
92:     RS_START = 4
93: } RecursiveIteratorState;
94:
95: typedef struct _spl_sub_iterator {
96:     zend_object_iterator *iterator;
97:     zval *zval;
98:     zend_class_entry *ce;
99:     RecursiveIteratorState state;
100: } spl_sub_iterator;
101:
102: typedef struct _spl_recursive_it_object {
103:     spl_sub_iterator *iterator;
104:     int level;
105:     RecursiveIteratorMode mode;
106:     int flags;
107:     int max_depth;
108:     zend_bool in_iteration;
109:     zend_function *beginIteration;
110:     zend_function *endIteration;
111:     zend_function *callHasChildren;
112:     zend_function *callGetChildren;
113:     zend_function *beginChildren;
114:     zend_function *endChildren;
115:     zend_function *nextElement;
116:     zend_class_entry *ce;
117:     smart_str prefix;
118:     smart_str postfix;
119:     zend_object *std;
120: } spl_recursive_it_object;
121:
122: typedef struct _spl_recursive_it_iterator {
123:     zend_object_iterator intern;
124:     spl_recursive_it_iterator;
125: }
126:
127: static zend_object_handlers spl_handlers_rec_it;
128: static zend_object_handlers spl_handlers_dual_it;
129:
130: static inline spl_recursive_it_object *spl_recursive_it_from_obj(zend_object *obj) /* {{{ */ {
131:     return (spl_recursive_it_object *) (char *) (obj) - XOFFSETOF(spl_recursive_it_object, std);
132: } /* }}} */
133:
134: #define SPL_RECURSIVE_IT_P(sv) spl_recursive_it_from_obj(Z_OBJ_P(sv))
135:
136: #define SPL_FETCH_AND_CHECK_DUAL_IT(var, objval) \
137:     do { \
138:         spl_dual_it_object *it = SPL_DUAL_IT_P(objval); \
139:         if (it->it_type == RIT_UNKNOWN) { \
140:             zend_throw_exception_ex(spl_ce_LogicException, 0, \
141:                 "The object is in an invalid state as the parent constructor was not called"); \
142:             return; \
143:         } \
144:         (var) = it; \
145:     } while (0)
146:
147: #define SPL_FETCH_SUB_ELEMENT(var, object, element) \
148:     do { \
149:         if (! (object->iterators)) { \
150:             zend_throw_exception_ex(spl_ce_LogicException, 0, \
151:                 "The object is in an invalid state as the parent constructor was not called"); \
152:             return; \
153:         } \
154:         (var) = (object->iterators)[object->level].element; \
155:     } while (0)
156:
157: #define SPL_FETCH_SUB_ELEMENT_ADOR(var, object, element) \
158:     do { \
159:         if (! (object->iterators)) { \
160:             zend_throw_exception_ex(spl_ce_LogicException, 0, \
161:                 "The object is in an invalid state as the parent constructor was not called"); \
162:             return; \
163:         } \
164:         (var) = (object->iterators)[object->level].element; \
165:     } while (0)
166:
167: #define SPL_FETCH_SUB_ITERATOR(var, object) SPL_FETCH_SUB_ELEMENT(var, object, iterator)
168:
169:
170: static void spl_recursive_it_ctor(zend_object_iterator *iter)
171: {
172:     spl_recursive_it_iterator *iter = (spl_recursive_it_iterator *) iter;
173:     spl_recursive_it_object *object = SPL_RECURSIVE_IT_P(iter->intern.data);
174:     zend_object_iterator *sub_iter;
175:
176:     while (object->level > 0) {
177:         if (!IS_UNDEF(object->iterators[object->level].obj)) {
178:             sub_iter = object->iterators[object->level].iterator;
179:             zend_iterator_dtor(sub_iter);
180:             zval_ptr_dtor(&object->iterators[object->level].obj);
181:         }
182:         object->level--;
183:     }
184:     object->iterators = zval_copy(&object->iterators, &object->sub_iterator);
185:     object->level = 0;
186:     zval_ptr_dtor(&iter->intern.data);
187: }
188:
189: static int spl_recursive_it_valid_obj(spl_recursive_it_object *object, zval *this)
190: {
191:     zend_object_iterator *sub_iter;
192:     int level = object->level;
193:
194:     if (! (object->iterators)) {
195:         return FAILURE;
196:     }
197:     while (level > 0) {
198:         sub_iter = object->iterators[level].iterator;
199:         if (sub_iter->funcs->valid(sub_iter) == SUCCESS) {
200:             return SUCCESS;
201:         }
202:         level--;
203:     }
204:     if (object->endIteration && object->in_iteration) {
205:         zend_call_method_with_0_params(this, object->ce, object->endIteration, "endIteration", NULL);
206:     }
207:     object->in_iteration = 0;
208:     return FAILURE;
209: }
210:
211: static int spl_recursive_it_valid(zend_object_iterator *iter)
212: {
213:     return spl_recursive_it_valid_obj(SPL_RECURSIVE_IT_P(iter->data), iter->data);
214: }
215:
216: static zval *spl_recursive_it_get_current_data(zend_object_iterator *iter)
217: {
218:     spl_recursive_it_object *object = SPL_RECURSIVE_IT_P(iter->data);
219:     zend_object_iterator *sub_iter = object->iterators[object->level].iterator;
220:     return sub_iter->funcs->get_current_data(sub_iter);
221: }
222:
223: static void spl_recursive_it_get_current_key(zend_object_iterator *iter, zval *key)
224: {
225:     spl_recursive_it_object *object = SPL_RECURSIVE_IT_P(iter->data);
226:     zend_object_iterator *sub_iter = object->iterators[object->level].iterator;
227:     if (sub_iter->funcs->get_current_key) {
228:         sub_iter->funcs->get_current_key(sub_iter, key);
229:     } else {
230:         ZVAL_LONG(key, iter->index);
231:     }
232: }
233:
234: static void spl_recursive_it_move_forward_obj(spl_recursive_it_object *object, zval *this)
235: {
236:     zend_object_iterator *iterator;
237:     zval *zval;
238:     zend_class_entry *ce;
239:     zval *retval;
240:     zend_object_iterator *sub_iter;
241:     int has_children;
242:
243:     SPL_FETCH_SUB_ITERATOR(iterator, object);
244:     while (!EG(exception)) {
245:         next_step:
246:         iterator = object->iterators[object->level].iterator;
247:         switch (object->iterators[object->level].state) {
248:             case RS_NEXT:
249:                 iterator->funcs->move_forward(iterator);
250:                 if (EG(exception)) {
251:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
252:                         return;
253:                     } else {
254:                         zend_clear_exception();
255:                     }
256:                 }
257:                 /* fall through */
258:             case RS_START:
259:                 if (iterator->funcs->valid(iterator) == FAILURE) {
260:                     break;
261:                 }
262:                 object->iterators[object->level].state = RS_TEST;
263:                 /* break; */
264:             case RS_TEST:
265:                 ce = object->iterators[object->level].ce;
266:                 sub_iter = object->iterators[object->level].sub_iter;
267:                 if (object->callHasChildren) {
268:                     zend_call_method_with_0_params(this, object->ce, object->callHasChildren, "callHasChildren", &retval);
269:                 } else {
270:                     zend_call_method_with_0_params(sub_iter, ce, NULL, "hasChildren", &retval);
271:                 }
272:                 if (EG(exception)) {
273:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
274:                         return;
275:                     }
276:                     object->iterators[object->level].state = RS_NEXT;
277:                     return;
278:                 } else {
279:                     zend_clear_exception();
280:                 }
281:                 break;
282:             case RS_CHILD:
283:                 if (IS_UNDEF(retval) != IS_UNDEF) {
284:                     has_children = zend_is_true(retval);
285:                     zval_ptr_dtor(&retval);
286:                 }
287:                 if (has_children) {
288:                     if (object->max_depth == -1 || object->max_depth > object->level) {
289:                         switch (object->mode) {
290:                             case RIT_LEAVES_ONLY:
291:                                 break;
292:                             case RIT_CHILD_FIRST:
293:                                 object->iterators[object->level].state = RS_CHILD;
294:                                 goto next_step;
295:                             case RIT_SELF_FIRST:
296:                                 object->iterators[object->level].state = RS_SELF;
297:                                 goto next_step;
298:                             default:
299:                                 break;
300:                         }
301:                     } else {
302:                         /* do not recurse into */
303:                         if (object->mode == RIT_LEAVES_ONLY) {
304:                             /* this is not a leave, so skip it */
305:                             object->iterators[object->level].state = RS_NEXT;
306:                             goto next_step;
307:                         }
308:                     }
309:                 }
310:                 if (object->nextElement) {
311:                     zend_call_method_with_0_params(this, object->ce, object->nextElement, "nextElement", NULL);
312:                 }
313:                 object->iterators[object->level].state = RS_NEXT;
314:                 if (object->flags & RIT_CATCH_GET_CHILD) {
315:                     return;
316:                 }
317:                 zend_clear_exception();
318:                 break;
319:             case RS_SELF:
320:                 return /* self */;
321:             case RS_SELF_FIRST:
322:                 if (object->nextElement && (object->mode == RIT_SELF_FIRST || object->mode == RIT_CHILD_FIRST)) {
323:                     zend_call_method_with_0_params(this, object->ce, object->nextElement, "nextElement", NULL);
324:                 }
325:                 if (object->mode == RIT_SELF_FIRST) {
326:                     object->iterators[object->level].state = RS_CHILD;
327:                 } else {
328:                     object->iterators[object->level].state = RS_NEXT;
329:                 }
330:                 return /* self */;
331:             case RS_CHILD:
332:                 ce = object->iterators[object->level].ce;
333:                 sub_iter = object->iterators[object->level].sub_iter;
334:                 if (object->callGetChildren) {
335:                     zend_call_method_with_0_params(this, object->ce, object->callGetChildren, "callGetChildren", &child);
336:                 } else {
337:                     zend_call_method_with_0_params(sub_iter, ce, NULL, "getChildren", &child);
338:                 }
339:                 if (EG(exception)) {
340:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
341:                         return;
342:                     }
343:                     zend_clear_exception();
344:                     zval_ptr_dtor(&child);
345:                     object->iterators[object->level].state = RS_NEXT;
346:                     goto next_step;
347:                 }
348:                 if (IS_UNDEF(child) != IS_UNDEF || IS_UNDEF(child) != IS_UNDEF) {
349:                     if (IS_UNDEF(child) && instanceof_function(ce, spl_ce_RecursiveIterator)) {
350:                         zval_ptr_dtor(&child);
351:                         zend_throw_exception(spl_ce_UnexpectedValueException, "Objects returned by RecursiveIterator::getChildren() must implement RecursiveIterator",
352:                             0);
353:                     }
354:                     return;
355:                 }
356:                 if (object->mode == RIT_CHILD_FIRST) {
357:                     object->iterators[object->level].state = RS_SELF;
358:                 } else {
359:                     object->iterators[object->level].state = RS_NEXT;
360:                 }
361:                 break;
362:             case RS_LEAVES_ONLY:
363:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
364:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
365:                 object->iterators[object->level].iterator = sub_iter;
366:                 object->iterators[object->level].ce = ce;
367:                 object->iterators[object->level].state = RS_START;
368:                 if (sub_iter->funcs->rewind) {
369:                     sub_iter->funcs->rewind(sub_iter);
370:                 }
371:                 if (object->beginChildren) {
372:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
373:                 }
374:                 if (EG(exception)) {
375:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
376:                         return;
377:                     }
378:                 }
379:                 break;
380:             case RS_SELF:
381:                 return;
382:             case RS_SELF_FIRST:
383:                 if (object->mode == RIT_CHILD_FIRST) {
384:                     object->iterators[object->level].state = RS_SELF;
385:                 } else {
386:                     object->iterators[object->level].state = RS_NEXT;
387:                 }
388:                 break;
389:             case RS_CHILD:
390:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
391:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
392:                 object->iterators[object->level].iterator = sub_iter;
393:                 object->iterators[object->level].ce = ce;
394:                 object->iterators[object->level].state = RS_START;
395:                 if (sub_iter->funcs->rewind) {
396:                     sub_iter->funcs->rewind(sub_iter);
397:                 }
398:                 if (object->beginChildren) {
399:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
400:                 }
401:                 if (EG(exception)) {
402:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
403:                         return;
404:                     }
405:                 }
406:                 break;
407:             case RS_SELF:
408:                 return;
409:             case RS_SELF_FIRST:
410:                 if (object->mode == RIT_CHILD_FIRST) {
411:                     object->iterators[object->level].state = RS_SELF;
412:                 } else {
413:                     object->iterators[object->level].state = RS_NEXT;
414:                 }
415:                 break;
416:             case RS_CHILD:
417:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
418:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
419:                 object->iterators[object->level].iterator = sub_iter;
420:                 object->iterators[object->level].ce = ce;
421:                 object->iterators[object->level].state = RS_START;
422:                 if (sub_iter->funcs->rewind) {
423:                     sub_iter->funcs->rewind(sub_iter);
424:                 }
425:                 if (object->beginChildren) {
426:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
427:                 }
428:                 if (EG(exception)) {
429:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
430:                         return;
431:                     }
432:                 }
433:                 break;
434:             case RS_SELF:
435:                 return;
436:             case RS_SELF_FIRST:
437:                 if (object->mode == RIT_CHILD_FIRST) {
438:                     object->iterators[object->level].state = RS_SELF;
439:                 } else {
440:                     object->iterators[object->level].state = RS_NEXT;
441:                 }
442:                 break;
443:             case RS_CHILD:
444:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
445:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
446:                 object->iterators[object->level].iterator = sub_iter;
447:                 object->iterators[object->level].ce = ce;
448:                 object->iterators[object->level].state = RS_START;
449:                 if (sub_iter->funcs->rewind) {
450:                     sub_iter->funcs->rewind(sub_iter);
451:                 }
452:                 if (object->beginChildren) {
453:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
454:                 }
455:                 if (EG(exception)) {
456:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
457:                         return;
458:                     }
459:                 }
460:                 break;
461:             case RS_SELF:
462:                 return;
463:             case RS_SELF_FIRST:
464:                 if (object->mode == RIT_CHILD_FIRST) {
465:                     object->iterators[object->level].state = RS_SELF;
466:                 } else {
467:                     object->iterators[object->level].state = RS_NEXT;
468:                 }
469:                 break;
470:             case RS_CHILD:
471:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
472:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
473:                 object->iterators[object->level].iterator = sub_iter;
474:                 object->iterators[object->level].ce = ce;
475:                 object->iterators[object->level].state = RS_START;
476:                 if (sub_iter->funcs->rewind) {
477:                     sub_iter->funcs->rewind(sub_iter);
478:                 }
479:                 if (object->beginChildren) {
480:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
481:                 }
482:                 if (EG(exception)) {
483:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
484:                         return;
485:                     }
486:                 }
487:                 break;
488:             case RS_SELF:
489:                 return;
490:             case RS_SELF_FIRST:
491:                 if (object->mode == RIT_CHILD_FIRST) {
492:                     object->iterators[object->level].state = RS_SELF;
493:                 } else {
494:                     object->iterators[object->level].state = RS_NEXT;
495:                 }
496:                 break;
497:             case RS_CHILD:
498:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
499:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
500:                 object->iterators[object->level].iterator = sub_iter;
501:                 object->iterators[object->level].ce = ce;
502:                 object->iterators[object->level].state = RS_START;
503:                 if (sub_iter->funcs->rewind) {
504:                     sub_iter->funcs->rewind(sub_iter);
505:                 }
506:                 if (object->beginChildren) {
507:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
508:                 }
509:                 if (EG(exception)) {
510:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
511:                         return;
512:                     }
513:                 }
514:                 break;
515:             case RS_SELF:
516:                 return;
517:             case RS_SELF_FIRST:
518:                 if (object->mode == RIT_CHILD_FIRST) {
519:                     object->iterators[object->level].state = RS_SELF;
520:                 } else {
521:                     object->iterators[object->level].state = RS_NEXT;
522:                 }
523:                 break;
524:             case RS_CHILD:
525:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
526:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
527:                 object->iterators[object->level].iterator = sub_iter;
528:                 object->iterators[object->level].ce = ce;
529:                 object->iterators[object->level].state = RS_START;
530:                 if (sub_iter->funcs->rewind) {
531:                     sub_iter->funcs->rewind(sub_iter);
532:                 }
533:                 if (object->beginChildren) {
534:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
535:                 }
536:                 if (EG(exception)) {
537:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
538:                         return;
539:                     }
540:                 }
541:                 break;
542:             case RS_SELF:
543:                 return;
544:             case RS_SELF_FIRST:
545:                 if (object->mode == RIT_CHILD_FIRST) {
546:                     object->iterators[object->level].state = RS_SELF;
547:                 } else {
548:                     object->iterators[object->level].state = RS_NEXT;
549:                 }
550:                 break;
551:             case RS_CHILD:
552:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
553:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
554:                 object->iterators[object->level].iterator = sub_iter;
555:                 object->iterators[object->level].ce = ce;
556:                 object->iterators[object->level].state = RS_START;
557:                 if (sub_iter->funcs->rewind) {
558:                     sub_iter->funcs->rewind(sub_iter);
559:                 }
560:                 if (object->beginChildren) {
561:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
562:                 }
563:                 if (EG(exception)) {
564:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
565:                         return;
566:                     }
567:                 }
568:                 break;
569:             case RS_SELF:
570:                 return;
571:             case RS_SELF_FIRST:
572:                 if (object->mode == RIT_CHILD_FIRST) {
573:                     object->iterators[object->level].state = RS_SELF;
574:                 } else {
575:                     object->iterators[object->level].state = RS_NEXT;
576:                 }
577:                 break;
578:             case RS_CHILD:
579:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
580:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
581:                 object->iterators[object->level].iterator = sub_iter;
582:                 object->iterators[object->level].ce = ce;
583:                 object->iterators[object->level].state = RS_START;
584:                 if (sub_iter->funcs->rewind) {
585:                     sub_iter->funcs->rewind(sub_iter);
586:                 }
587:                 if (object->beginChildren) {
588:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
589:                 }
590:                 if (EG(exception)) {
591:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
592:                         return;
593:                     }
594:                 }
595:                 break;
596:             case RS_SELF:
597:                 return;
598:             case RS_SELF_FIRST:
599:                 if (object->mode == RIT_CHILD_FIRST) {
600:                     object->iterators[object->level].state = RS_SELF;
601:                 } else {
602:                     object->iterators[object->level].state = RS_NEXT;
603:                 }
604:                 break;
605:             case RS_CHILD:
606:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
607:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
608:                 object->iterators[object->level].iterator = sub_iter;
609:                 object->iterators[object->level].ce = ce;
610:                 object->iterators[object->level].state = RS_START;
611:                 if (sub_iter->funcs->rewind) {
612:                     sub_iter->funcs->rewind(sub_iter);
613:                 }
614:                 if (object->beginChildren) {
615:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
616:                 }
617:                 if (EG(exception)) {
618:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
619:                         return;
620:                     }
621:                 }
622:                 break;
623:             case RS_SELF:
624:                 return;
625:             case RS_SELF_FIRST:
626:                 if (object->mode == RIT_CHILD_FIRST) {
627:                     object->iterators[object->level].state = RS_SELF;
628:                 } else {
629:                     object->iterators[object->level].state = RS_NEXT;
630:                 }
631:                 break;
632:             case RS_CHILD:
633:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
634:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
635:                 object->iterators[object->level].iterator = sub_iter;
636:                 object->iterators[object->level].ce = ce;
637:                 object->iterators[object->level].state = RS_START;
638:                 if (sub_iter->funcs->rewind) {
639:                     sub_iter->funcs->rewind(sub_iter);
640:                 }
641:                 if (object->beginChildren) {
642:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
643:                 }
644:                 if (EG(exception)) {
645:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
646:                         return;
647:                     }
648:                 }
649:                 break;
650:             case RS_SELF:
651:                 return;
652:             case RS_SELF_FIRST:
653:                 if (object->mode == RIT_CHILD_FIRST) {
654:                     object->iterators[object->level].state = RS_SELF;
655:                 } else {
656:                     object->iterators[object->level].state = RS_NEXT;
657:                 }
658:                 break;
659:             case RS_CHILD:
660:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
661:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
662:                 object->iterators[object->level].iterator = sub_iter;
663:                 object->iterators[object->level].ce = ce;
664:                 object->iterators[object->level].state = RS_START;
665:                 if (sub_iter->funcs->rewind) {
666:                     sub_iter->funcs->rewind(sub_iter);
667:                 }
668:                 if (object->beginChildren) {
669:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
670:                 }
671:                 if (EG(exception)) {
672:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
673:                         return;
674:                     }
675:                 }
676:                 break;
677:             case RS_SELF:
678:                 return;
679:             case RS_SELF_FIRST:
680:                 if (object->mode == RIT_CHILD_FIRST) {
681:                     object->iterators[object->level].state = RS_SELF;
682:                 } else {
683:                     object->iterators[object->level].state = RS_NEXT;
684:                 }
685:                 break;
686:             case RS_CHILD:
687:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
688:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
689:                 object->iterators[object->level].iterator = sub_iter;
690:                 object->iterators[object->level].ce = ce;
691:                 object->iterators[object->level].state = RS_START;
692:                 if (sub_iter->funcs->rewind) {
693:                     sub_iter->funcs->rewind(sub_iter);
694:                 }
695:                 if (object->beginChildren) {
696:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
697:                 }
698:                 if (EG(exception)) {
699:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
700:                         return;
701:                     }
702:                 }
703:                 break;
704:             case RS_SELF:
705:                 return;
706:             case RS_SELF_FIRST:
707:                 if (object->mode == RIT_CHILD_FIRST) {
708:                     object->iterators[object->level].state = RS_SELF;
709:                 } else {
710:                     object->iterators[object->level].state = RS_NEXT;
711:                 }
712:                 break;
713:             case RS_CHILD:
714:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
715:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
716:                 object->iterators[object->level].iterator = sub_iter;
717:                 object->iterators[object->level].ce = ce;
718:                 object->iterators[object->level].state = RS_START;
719:                 if (sub_iter->funcs->rewind) {
720:                     sub_iter->funcs->rewind(sub_iter);
721:                 }
722:                 if (object->beginChildren) {
723:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
724:                 }
725:                 if (EG(exception)) {
726:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
727:                         return;
728:                     }
729:                 }
730:                 break;
731:             case RS_SELF:
732:                 return;
733:             case RS_SELF_FIRST:
734:                 if (object->mode == RIT_CHILD_FIRST) {
735:                     object->iterators[object->level].state = RS_SELF;
736:                 } else {
737:                     object->iterators[object->level].state = RS_NEXT;
738:                 }
739:                 break;
740:             case RS_CHILD:
741:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
742:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
743:                 object->iterators[object->level].iterator = sub_iter;
744:                 object->iterators[object->level].ce = ce;
745:                 object->iterators[object->level].state = RS_START;
746:                 if (sub_iter->funcs->rewind) {
747:                     sub_iter->funcs->rewind(sub_iter);
748:                 }
749:                 if (object->beginChildren) {
750:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
751:                 }
752:                 if (EG(exception)) {
753:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
754:                         return;
755:                     }
756:                 }
757:                 break;
758:             case RS_SELF:
759:                 return;
760:             case RS_SELF_FIRST:
761:                 if (object->mode == RIT_CHILD_FIRST) {
762:                     object->iterators[object->level].state = RS_SELF;
763:                 } else {
764:                     object->iterators[object->level].state = RS_NEXT;
765:                 }
766:                 break;
767:             case RS_CHILD:
768:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
769:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
770:                 object->iterators[object->level].iterator = sub_iter;
771:                 object->iterators[object->level].ce = ce;
772:                 object->iterators[object->level].state = RS_START;
773:                 if (sub_iter->funcs->rewind) {
774:                     sub_iter->funcs->rewind(sub_iter);
775:                 }
776:                 if (object->beginChildren) {
777:                     zend_call_method_with_0_params(this, object->ce, object->beginChildren, "beginChildren", NULL);
778:                 }
779:                 if (EG(exception)) {
780:                     if (! (object->flags & RIT_CATCH_GET_CHILD)) {
781:                         return;
782:                     }
783:                 }
784:                 break;
785:             case RS_SELF:
786:                 return;
787:             case RS_SELF_FIRST:
788:                 if (object->mode == RIT_CHILD_FIRST) {
789:                     object->iterators[object->level].state = RS_SELF;
790:                 } else {
791:                     object->iterators[object->level].state = RS_NEXT;
792:                 }
793:                 break;
794:             case RS_CHILD:
795:                 object->iterators = zval_copy(&object->iterators, &object->sub_iterator) /* ++(object->level+1) */;
796:                 ZVAL_COPY_VALUE(object->iterators[object->level].obj, child);
797:                 object->iterators[object->level].
```

```

376:     } else {
377:         zend_clear_exception();
378:     }
379: }
380: }
381: goto next_step;
382: }
383: /* no more elements */
384: if (object->level > 0) {
385:     if (object->endChildren) {
386:         zend_call_method_with_0_params(&this, object->ce, object->zendChildren, "endchildren", NULL);
387:     }
388:     if (EG(exception)) {
389:         if (!object->flags & RIT_CATCH_GET_CHILD) {
390:             return;
391:         } else {
392:             zend_clear_exception();
393:         }
394:     }
395:     if (object->level > 0) {
396:         zval garbage;
397:         ZVAL_COPY_VALUE(&garbage, sub_iter->iterators[object->level].sub_iter);
398:         ZVAL_UNDEF(&object->iterators[object->level].sub_iter);
399:         zval_ptr_dtor(&garbage);
400:         zend_iterator_dtor(iterator);
401:         object->level--;
402:     }
403: } else {
404:     return; /* done completely */
405: }
406: }
407: }
408:
409: static void spl_recursive_it_rewind_ex(spl_recursive_it_object *object, zval *rthis)
410: {
411:     zend_object_iterator *sub_iter;
412:
413:     SPL_FETCH_SUB_ITERATOR(sub_iter, object);
414:
415:     while (object->level) {
416:         sub_iter = object->iterators[object->level].iterator;
417:         zend_iterator_dtor(sub_iter);
418:         zval_ptr_dtor(&object->iterators[object->level-1].sub_iter);
419:         if (EG(exception) && (object->endChildren || object->endChildren->common.scope != spl_ce_RecursiveIteratorIterator)) {
420:             zend_call_method_with_0_params(&this, object->ce, object->zendChildren, "endchildren", NULL);
421:         }
422:     }
423:     object->iterators = erealloc(object->iterators, sizeof(spl_sub_iterator));
424:     object->iterators[0].state = RS_START;
425:     sub_iter = object->iterators[0].iterator;
426:     if (sub_iter->funcs->rewind) {
427:         sub_iter->funcs->rewind(sub_iter);
428:     }
429:     if (EG(exception) && object->beginIteration == 1) {
430:         zend_call_method_with_0_params(&this, object->ce, object->beginIteration, "beginIteration", NULL);
431:     }
432:     object->in_iteration = 1;
433:     spl_recursive_it_move_forward_ex(object, rthis);
434: }
435:
436: static void spl_recursive_it_move_forward(spl_recursive_it_object *iter)
437: {
438:     spl_recursive_it_move_forward_ex(&iter->sub_iter, iter->data);
439: }
440:
441: static void spl_recursive_it_rewind(spl_recursive_it_object *iter)
442: {
443:     spl_recursive_it_rewind_ex(&iter->sub_iter, iter->data);
444: }
445:
446: static zend_object_iterator *spl_recursive_it_get_iterator(spl_recursive_it_object *ce, zval *object, int by_ref)
447: {
448:     spl_recursive_it_iterator *iterator;
449:     spl_recursive_it_object *obj;
450:
451:     if (by_ref) {
452:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
453:         return NULL;
454:     }
455:     iterator = emalloc(sizeof(spl_recursive_it_iterator));
456:     object = &iter->sub_iter->it_P(object);
457:     if (object->iterators == NULL) {
458:         zend_error(E_ERROR, "The object to be iterated is in an invalid state: "
459:             "the parent constructor has not been called");
460:     }
461:
462:     zend_iterator_init(&zend_object_iterator* iterator);
463:
464:     ZVAL_COPY(&iterator->intern.data, object);
465:     iterator->intern.funcs = ce->iterator_funcs.funcs;
466:     return (&zend_object_iterator* iterator);
467: }
468:
469: static const zend_object_iterator_funcs spl_recursive_it_iterator_funcs = {
470:     spl_recursive_it_dtor,
471:     spl_recursive_it_valid,
472:     spl_recursive_it_get_current_data,
473:     spl_recursive_it_get_current_key,
474:     spl_recursive_it_move_forward,
475:     spl_recursive_it_rewind,
476:     NULL
477: };
478:
479: static void spl_recursive_it_it_construct(INTERNAL_FUNCTION_PARAMETERS, zend_class_entry *ce_base, zend_class_entry *ce_inner, recursive_it_type rit_type)
480: {
481:     zval *object = getThis();
482:     spl_recursive_it_object *intern;
483:     zval *iterator;
484:     zend_class_entry *ce_iterator;
485:     zend_long mode, flags;
486:     zend_error_handling error_handling;
487:     zval caching_it, aggregate_retval;
488:
489:     zend_replace_error_handling(EH_THROW, spl_ce_InvalidArgumentException, &error_handling);
490:
491:     switch (rit_type) {
492:         case RIT_RecursiveForAggregate: {
493:             zval caching_it_flags, *user_caching_it_flags = NULL;
494:             mode = RIT_HELP_FIRST;
495:             flags = RIT_BYPASS_RIT;
496:
497:             if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "o|l", iterator, &mode, &flags, user_caching_it_flags, &mode) == SUCCESS) {
498:                 if (instanceof_function(&OBJECT_P(iterator), &ce_base, &ce_iterator)) {
499:                     zend_call_method_with_0_params(iterator, &OBJECT_P(iterator), &OBJECT_P(iterator)->iterator_funcs.if_new_iterator, "getIterator", &aggregate_retval);
500:
501:                     iterator = &aggregate_retval;
502:                 } else {
503:                     Z_ADDREF_P(iterator);
504:                 }
505:
506:                 if (user_caching_it_flags) {
507:                     ZVAL_COPY(&caching_it_flags, user_caching_it_flags);
508:                 } else {
509:                     ZVAL_LONG(&caching_it_flags, RIT_CATCH_GET_CHILD);
510:                 }
511:                 spl_instantiate_arg_ex2(spl_ce_RecursiveCachingIterator, &caching_it, iterator, &caching_it_flags);
512:                 zval_ptr_dtor(&caching_it_flags);
513:                 zval_ptr_dtor(iterator);
514:                 iterator = &caching_it;
515:             } else {
516:                 iterator = NULL;
517:             }
518:             break;
519:         }
520:         case RIT_RecursiveIteratorIterator: {
521:             default: {
522:                 mode = RIT_LEAVES_ONLY;
523:                 flags = 0;
524:
525:                 if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "o|l", iterator, &mode, &flags) == SUCCESS) {
526:                     if (instanceof_function(&OBJECT_P(iterator), &ce_base, &ce_iterator)) {
527:                         zend_call_method_with_0_params(iterator, &OBJECT_P(iterator), &OBJECT_P(iterator)->iterator_funcs.if_new_iterator, "getIterator", &aggregate_retval);
528:
529:                         iterator = &aggregate_retval;
530:                     } else {
531:                         Z_ADDREF_P(iterator);
532:                     }
533:                 } else {
534:                     iterator = NULL;
535:                 }
536:                 break;
537:             }
538:         }
539:         if (iterator) {
540:             zval_ptr_dtor(iterator);
541:         }
542:         zend_throw_exception(spl_ce_InvalidArgumentException, "An instance of RecursiveIterator or IteratorAggregate creating it is required", 0);
543:         zend_restore_error_handling(&error_handling);
544:         return;
545:     }
546:
547:     intern = &SPL_RECURSIVE_IT_P(object);
548:     intern->iterators = emalloc(sizeof(spl_sub_iterator));
549:     intern->level = 0;
550:     intern->mode = mode;
551:     intern->flags = (int) flags;
552:     intern->max_depth = -1;
553:     intern->in_iteration = 0;
554:     intern->ce = &OBJECT_P(object);
555:
556:     intern->beginIteration = zend_hash_str_find_ptr(intern->ce->function_table, "beginIteration", sizeof("beginIteration") - 1);
557:     if (intern->beginIteration->common.scope == ce_base) {
558:         intern->beginIteration = NULL;
559:     }
560:     intern->endIteration = zend_hash_str_find_ptr(intern->ce->function_table, "endIteration", sizeof("endIteration") - 1);
561:     if (intern->endIteration->common.scope == ce_base) {
562:         intern->endIteration = NULL;
563:     }
564:     intern->callHasChildren = zend_hash_str_find_ptr(intern->ce->function_table, "callHasChildren", sizeof("callHasChildren") - 1);
565:     if (intern->callHasChildren->common.scope == ce_base) {
566:         intern->callHasChildren = NULL;
567:     }
568:     intern->callGetChildren = zend_hash_str_find_ptr(intern->ce->function_table, "callGetChildren", sizeof("callGetChildren") - 1);
569:     if (intern->callGetChildren->common.scope == ce_base) {
570:         intern->callGetChildren = NULL;
571:     }
572:     intern->beginChildren = zend_hash_str_find_ptr(intern->ce->function_table, "beginChildren", sizeof("beginChildren") - 1);
573:     if (intern->beginChildren->common.scope == ce_base) {
574:         intern->beginChildren = NULL;
575:     }
576:     intern->endChildren = zend_hash_str_find_ptr(intern->ce->function_table, "endChildren", sizeof("endChildren") - 1);
577:     if (intern->endChildren->common.scope == ce_base) {
578:         intern->endChildren = NULL;
579:     }
580:     intern->nextElement = zend_hash_str_find_ptr(intern->ce->function_table, "nextElement", sizeof("nextElement") - 1);
581:     if (intern->nextElement->common.scope == ce_base) {
582:         intern->nextElement = NULL;
583:     }
584:
585:     ce_iterator = &OBJECT_P(iterator); /* respect inheritance, don't use spl_ce_RecursiveIterator */
586:     intern->iterators[0].iterator = ce_iterator->get_iterator(ce_iterator, iterator, 0);
587:     ZVAL_COPY_VALUE(intern->iterators[0].sub_iter, iterator);
588:     intern->iterators[0].ce = ce_iterator;
589:     intern->iterators[0].state = RS_START;
590:
591:     zend_restore_error_handling(&error_handling);
592:
593:     if (EG(exception)) {
594:         zend_object_iterator *sub_iter;
595:
596:         while (intern->level > 0) {
597:             sub_iter = intern->iterators[intern->level].iterator;
598:             zend_iterator_dtor(sub_iter);
599:             zval_ptr_dtor(&intern->iterators[intern->level-1].sub_iter);
600:         }
601:         ifree(intern->iterators);
602:         intern->iterators = NULL;
603:     }
604: }
605:
606: /* {{{ proto void RecursiveIteratorIterator::__construct(RecursiveIteratorIteratorAggregate $it, $int mode = RIT_LEAVES_ONLY, $int flags = 0) throw $InvalidArgumentException
607: * Creates a RecursiveIteratorIterator from a RecursiveIterator. */
608: SPL_METHOD(RecursiveIteratorIterator, __construct)
609: {
610:     spl_recursive_it_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveIteratorIterator, &zend_object_iterator, RIT_RecursiveIteratorIterator);
611: } /* }}} */
612:
613: /* {{{ proto void RecursiveIteratorIterator::rewind()
614: * Rewind the iterator to the first element of the top level inner iterator. */
615: SPL_METHOD(RecursiveIteratorIterator, rewind)
616: {
617:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
618:
619:     if (zend_parse_parameters_none() == FAILURE) {
620:         return;
621:     }
622:
623:     spl_recursive_it_rewind_ex(object, getThis());
624: } /* }}} */
625:
626: /* {{{ proto bool RecursiveIteratorIterator::valid()
627: * Check whether the current position is valid */
628: SPL_METHOD(RecursiveIteratorIterator, valid)
629: {
630:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
631:
632:     if (zend_parse_parameters_none() == FAILURE) {
633:         return;
634:     }
635:
636:     RETURN_BOOL(spl_recursive_it_valid_ex(object, getThis()) == SUCCESS);
637: } /* }}} */
638:
639: /* {{{ proto mixed RecursiveIteratorIterator::key()
640: * Access the current key */
641: SPL_METHOD(RecursiveIteratorIterator, key)
642: {
643:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
644:     zend_object_iterator *iterator;
645:
646:     if (zend_parse_parameters_none() == FAILURE) {
647:         return;
648:     }
649:
650:     SPL_FETCH_SUB_ITERATOR(iterator, object);
651:
652:     if (iterator->funcs->get_current_key) {
653:         iterator->funcs->get_current_key(iterator, &return_value);
654:     } else {
655:         RETURN_NULL();
656:     }
657: } /* }}} */
658:
659: /* {{{ proto mixed RecursiveIteratorIterator::current()
660: * Access the current element value */
661: SPL_METHOD(RecursiveIteratorIterator, current)
662: {
663:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
664:     zend_object_iterator *iterator;
665:     zval *data;
666:
667:     if (zend_parse_parameters_none() == FAILURE) {
668:         return;
669:     }
670:
671:     SPL_FETCH_SUB_ITERATOR(iterator, object);
672:
673:     data = iterator->funcs->get_current_data(iterator);
674:     if (data) {
675:         ZVAL_DEREF(data);
676:         ZVAL_COPY(&return_value, data);
677:     }
678: } /* }}} */
679:
680: /* {{{ proto void RecursiveIteratorIterator::next()
681: * Move forward to the next element */
682: SPL_METHOD(RecursiveIteratorIterator, next)
683: {
684:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
685:
686:     if (zend_parse_parameters_none() == FAILURE) {
687:         return;
688:     }
689:
690:     spl_recursive_it_move_forward_ex(object, getThis());
691: } /* }}} */
692:
693: /* {{{ proto int RecursiveIteratorIterator::getDepth()
694: * Get the current depth of the recursive iteration */
695: SPL_METHOD(RecursiveIteratorIterator, getDepth)
696: {
697:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
698:
699:     if (zend_parse_parameters_none() == FAILURE) {
700:         return;
701:     }
702:
703:     RETURN_LONG(object->level);
704: } /* }}} */
705:
706: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::getSubIterator($int level)
707: * The current active sub iterator or the iterator at specified level */
708: SPL_METHOD(RecursiveIteratorIterator, getSubIterator)
709: {
710:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
711:     zend_long level = object->level;
712:     zval *value;
713:
714:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "[i]", &level) == FAILURE) {
715:         return;
716:     }
717:     if (level < 0 || level > object->level) {
718:         RETURN_NULL();
719:     }
720:
721:     if (object->iterators) {
722:         zend_throw_exception(spl_ce_LogicException, 0,
723:             "The object is in an invalid state as the parent constructor was not called");
724:         return;
725:     }
726:
727:     value = object->iterators[level].sub_iter;
728:     ZVAL_DEREF(value);
729:     ZVAL_COPY(&return_value, value);
730: } /* }}} */
731:
732: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::getInnerIterator()
733: * The current active sub iterator */
734: SPL_METHOD(RecursiveIteratorIterator, getInnerIterator)
735: {
736:     spl_recursive_it_object *object = &SPL_RECURSIVE_IT_P(getThis());
737:     zval *sub_iter;
738:
739:     if (zend_parse_parameters_none() == FAILURE) {
740:         return;
741:     }
742:
743:     SPL_FETCH_SUB_ELEMENT_ADDR(sub_iter, object, sub_iter);
744:
745:     ZVAL_DEREF(sub_iter);
746:     ZVAL_COPY(&return_value, sub_iter);
747: } /* }}} */

```

```

748: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::beginIteration()
749: Called when iteration begins (after first rewind() call) */
750: SPL_METHOD(RecursiveIteratorIterator, beginIteration)
751: {
752:     IF (zend_parse_parameters_none() == FAILURE) {
753:         return;
754:     }
755:     /* nothing to do */
756:     /* }}} */
757: }
758:
759: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::endIteration()
760: Called when iteration ends (when valid() first returns false) */
761: SPL_METHOD(RecursiveIteratorIterator, endIteration)
762: {
763:     IF (zend_parse_parameters_none() == FAILURE) {
764:         return;
765:     }
766:     /* nothing to do */
767:     /* }}} */
768: }
769: /* {{{ proto bool RecursiveIteratorIterator::callHasChildren()
770: Called for each element to test whether it has children */
771: SPL_METHOD(RecursiveIteratorIterator, callHasChildren)
772: {
773:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
774:     zend_class_entry *ce;
775:     zval *obj;
776:
777:     IF (zend_parse_parameters_none() == FAILURE) {
778:         return;
779:     }
780:
781:     IF (obj->iterators) {
782:         RETURN_NULL();
783:     }
784:
785:     SPL_FETCH_SUB_ELEMENT(ce, object, ce);
786:
787:     obj = obj->iterators[object->level].obj;
788:     IF (Z_TYPE_P(obj) == IS_UNDEF) {
789:         RETURN_FALSE;
790:     } else {
791:         zend_call_method_with_0_params(obj, ce, NULL, "haschildren", return_value);
792:         IF (Z_TYPE_P(return_value) == IS_UNDEF) {
793:             RETURN_FALSE;
794:         }
795:     }
796:     /* }}} */
797: }
798: /* {{{ proto RecursiveIterator RecursiveIteratorIterator::callGetChildren()
799: Return children of current element */
800: SPL_METHOD(RecursiveIteratorIterator, callGetChildren)
801: {
802:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
803:     zend_class_entry *ce;
804:     zval *obj;
805:
806:     IF (zend_parse_parameters_none() == FAILURE) {
807:         return;
808:     }
809:
810:     SPL_FETCH_SUB_ELEMENT(ce, object, ce);
811:
812:     obj = obj->iterators[object->level].obj;
813:     IF (Z_TYPE_P(obj) == IS_UNDEF) {
814:         return;
815:     } else {
816:         zend_call_method_with_0_params(obj, ce, NULL, "getchildren", return_value);
817:         IF (Z_TYPE_P(return_value) == IS_UNDEF) {
818:             RETURN_NULL();
819:         }
820:     }
821:     /* }}} */
822: }
823: /* {{{ proto void RecursiveIteratorIterator::beginChildren()
824: Called when recursing one level down */
825: SPL_METHOD(RecursiveIteratorIterator, beginChildren)
826: {
827:     IF (zend_parse_parameters_none() == FAILURE) {
828:         return;
829:     }
830:     /* nothing to do */
831:     /* }}} */
832: }
833: /* {{{ proto void RecursiveIteratorIterator::endChildren()
834: Called when end recursing one level */
835: SPL_METHOD(RecursiveIteratorIterator, endChildren)
836: {
837:     IF (zend_parse_parameters_none() == FAILURE) {
838:         return;
839:     }
840:     /* nothing to do */
841:     /* }}} */
842: }
833: /* {{{ proto void RecursiveIteratorIterator::nextElement()
834: Called when the next element is available */
845: SPL_METHOD(RecursiveIteratorIterator, nextElement)
846: {
847:     IF (zend_parse_parameters_none() == FAILURE) {
848:         return;
849:     }
850:     /* nothing to do */
851:     /* }}} */
852: }
853: /* {{{ proto void RecursiveIteratorIterator::setMaxDepth([max_depth = -1])
854: Set the maximum allowed depth (or any depth if max_depth = -1) */
855: SPL_METHOD(RecursiveIteratorIterator, setMaxDepth)
856: {
857:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
858:     zend_long max_depth = -1;
859:
860:     IF (zend_parse_parameters(ZEND_NUM_ARGS(), "l", &max_depth) == FAILURE) {
861:         return;
862:     }
863:     IF (max_depth < -1) {
864:         zend_throw_exception(spl_ce_OutOfRangeException, "Parameter max_depth must be >= -1", 0);
865:         return;
866:     } else IF (max_depth > INT_MAX) {
867:         max_depth = INT_MAX;
868:     }
869:
870:     object->max_depth = (int)max_depth;
871:     /* }}} */
872: }
873: /* {{{ proto int|false RecursiveIteratorIterator::getMaxDepth()
874: Return the maximum accepted depth or false if any depth is allowed */
875: SPL_METHOD(RecursiveIteratorIterator, getMaxDepth)
876: {
877:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
878:
879:     IF (zend_parse_parameters_none() == FAILURE) {
880:         return;
881:     }
882:
883:     IF (object->max_depth == -1) {
884:         RETURN_FALSE;
885:     } else {
886:         RETURN_LONG(object->max_depth);
887:     }
888:     /* }}} */
889: }
890:
891: static union zend_function *spl_recursive_it_get_method(zend_object **obj, zend_string *method, const zval *key)
892: {
893:     union_zend_function *function_handler;
894:     spl_recursive_it_object *object = spl_recursive_it_from_obj(*obj);
895:     zend_long level = object->level;
896:     zval *obj;
897:
898:     IF (obj->iterators) {
899:         php_error_docref(NULL, E_WARNING, "The to instance wasn't initialized properly", ZSTR_VAL(*obj), "ce->name");
900:     }
901:     obj = obj->iterators[level].obj;
902:     function_handler = std_object_handlers.get_method(obj, method, key);
903:     IF (function_handler) {
904:         IF (function_handler == zend_hash_find_ptr(&Z_OBJ_P(obj)->function_table, method) == NULL) {
905:             IF (Z_OBJ_HT_P(obj)->get_method) {
906:                 *obj = Z_OBJ_P(obj);
907:                 function_handler = (*obj->handlers)->get_method(obj, method, key);
908:             } else {
909:                 *obj = Z_OBJ_P(obj);
910:             }
911:         }
912:     }
913:     return function_handler;
914: }
915:
916: /* {{{ spl_recursive_iterator_iterator_ctor */
917: static void spl_recursive_iterator_iterator_ctor(zend_object **obj)
918: {
919:     spl_recursive_it_object *object = spl_recursive_it_from_obj(*obj);
920:     zend_object_iterator *sub_iter;
921:
922:     /* call standard ctor */
923:     zend_objects_destroy_object(*obj);
924:
925:     IF (obj->iterators) {
926:         while (object->level >= 0) {
927:             sub_iter = obj->iterators[object->level].iterator;
928:             zend_iterator_ctor(sub_iter);
929:             zval_ptr_dtor(&obj->iterators[object->level--].obj);
930:         }
931:     }
932:     obj->iterators = NULL;
933: }
934:
935: /* }}} */

```

```

1124: smart_str_free(object->prefix[part]);
1125:
1126: smart_str_append(object->prefix[part], prefix, prefix_len);
1127: /* '}' */
1128:
1129: /* {{{ proto string RecursiveTreeIterator::getPrefix()
1130:  Returns the string to place in front of current element */
1131: SPL_METHOD (RecursiveTreeIterator, getPrefix)
1132: {
1133:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1134:
1135:     if (zend_parse_parameters_none() == FAILURE) {
1136:         return;
1137:     }
1138:
1139:     if(object->it-iterators) {
1140:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1141:             "The object is in an invalid state as the parent constructor was not called");
1142:         return;
1143:     }
1144:
1145:     spl_recursive_tree_iterator_get_prefix(object, return_value);
1146:     /* '}' */
1147:
1148:     /* {{{ proto void RecursiveTreeIterator::setPostfix(string prefix)
1149:      Sets postfix as used in getPrefix() */
1150: SPL_METHOD (RecursiveTreeIterator, setPostfix)
1151: {
1152:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1153:     char* postfix;
1154:     size_t postfix_len;
1155:
1156:     if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s", &postfix, &postfix_len) == FAILURE) {
1157:         return;
1158:     }
1159:
1160:     smart_str_free(object->postfix[0]);
1161:     smart_str_append(object->postfix[0], postfix, postfix_len);
1162:     /* '}' */
1163:
1164:     /* {{{ proto string RecursiveTreeIterator::getEntry()
1165:  Returns the string presentation built for current element */
1166: SPL_METHOD (RecursiveTreeIterator, getEntry)
1167: {
1168:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1169:
1170:     if (zend_parse_parameters_none() == FAILURE) {
1171:         return;
1172:     }
1173:
1174:     if(object->it-iterators) {
1175:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1176:             "The object is in an invalid state as the parent constructor was not called");
1177:         return;
1178:     }
1179:
1180:     spl_recursive_tree_iterator_get_entry(object, return_value);
1181:     /* '}' */
1182:
1183:     /* {{{ proto string RecursiveTreeIterator::getPostfix()
1184:  Returns the string to place after the current element */
1185: SPL_METHOD (RecursiveTreeIterator, getPostfix)
1186: {
1187:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1188:
1189:     if (zend_parse_parameters_none() == FAILURE) {
1190:         return;
1191:     }
1192:
1193:     if(object->it-iterators) {
1194:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1195:             "The object is in an invalid state as the parent constructor was not called");
1196:         return;
1197:     }
1198:
1199:     spl_recursive_tree_iterator_get_postfix(object, return_value);
1200:     /* '}' */
1201:
1202:     /* {{{ proto mixed RecursiveTreeIterator::current()
1203:  Returns the current element prefixed and postfixed */
1204: SPL_METHOD (RecursiveTreeIterator, current)
1205: {
1206:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1207:     zval prefix, entry, postfix;
1208:     char *ptr;
1209:     zend_string *str;
1210:
1211:     if (zend_parse_parameters_none() == FAILURE) {
1212:         return;
1213:     }
1214:
1215:     if(object->it-iterators) {
1216:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1217:             "The object is in an invalid state as the parent constructor was not called");
1218:         return;
1219:     }
1220:
1221:     if (object->flags & BIT1_BYPASS_CURRENT) {
1222:         zend_object_iterator *iterator = object->it-iterators[object->level].iterator;
1223:         zval *data;
1224:
1225:         SPL_FETCH_SUB_ITERATOR(iterator, object);
1226:         data = iterator->funcs->get_current_data(iterator);
1227:         if (data) {
1228:             ZVAL_DEREF(data);
1229:             ZVAL_COPY(return_value, data);
1230:             return;
1231:         } else {
1232:             RETURN_NULL();
1233:         }
1234:     }
1235:
1236:     ZVAL_NULL(&prefix);
1237:     ZVAL_NULL(&entry);
1238:     spl_recursive_tree_iterator_get_prefix(object, &prefix);
1239:     spl_recursive_tree_iterator_get_entry(object, &entry);
1240:     if (Z_TYPE(entry) != IS_STRING) {
1241:         zval_ptr_dtor(&entry);
1242:         zval_ptr_dtor(&prefix);
1243:         RETURN_NULL();
1244:     }
1245:
1246:     spl_recursive_tree_iterator_get_postfix(object, &postfix);
1247:
1248:     str = zend_string_alloc(Z_STRLEN(prefix) + Z_STRLEN(entry) + Z_STRLEN(postfix), 0);
1249:     ptr = ZSTR_VAL(str);
1250:
1251:     memcpy(ptr, Z_STRVAL(prefix), Z_STRLEN(prefix));
1252:     ptr += Z_STRLEN(prefix);
1253:     memcpy(ptr, Z_STRVAL(entry), Z_STRLEN(entry));
1254:     ptr += Z_STRLEN(entry);
1255:     memcpy(ptr, Z_STRVAL(postfix), Z_STRLEN(postfix));
1256:     ptr += Z_STRLEN(postfix);
1257:     *ptr = 0;
1258:
1259:     zval_ptr_dtor(&prefix);
1260:     zval_ptr_dtor(&entry);
1261:     zval_ptr_dtor(&postfix);
1262:
1263:     RETURN_NEW_STR(str);
1264:     /* '}' */
1265:
1266:     /* {{{ proto mixed RecursiveTreeIterator::key()
1267:  Returns the current key prefixed and postfixed */
1268: SPL_METHOD (RecursiveTreeIterator, key)
1269: {
1270:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1271:     zend_object_iterator *iterator;
1272:     zval prefix, key, postfix, key_copy;
1273:     char *ptr;
1274:     zend_string *str;
1275:
1276:     if (zend_parse_parameters_none() == FAILURE) {
1277:         return;
1278:     }
1279:
1280:     SPL_FETCH_SUB_ITERATOR(iterator, object);
1281:
1282:     if (iterator->funcs->get_current_key() {
1283:         iterator->funcs->get_current_key(iterator, &key);
1284:     } else {
1285:         ZVAL_NULL(&key);
1286:     }
1287:
1288:     if (object->flags & BIT1_BYPASS_KEY) {
1289:         RETVAL_ZVAL(&key, 1, 1);
1290:         return;
1291:     }
1292:
1293:     if (Z_TYPE(key) != IS_STRING) {
1294:         if (zend_make_printable_zval(&key, &key_copy)) {
1295:             key = key_copy;
1296:         }
1297:     }
1298:
1299:     spl_recursive_tree_iterator_get_prefix(object, &prefix);
1300:     spl_recursive_tree_iterator_get_postfix(object, &postfix);
1301:
1302:     str = zend_string_alloc(Z_STRLEN(prefix) + Z_STRLEN(key) + Z_STRLEN(postfix), 0);
1303:     ptr = ZSTR_VAL(str);
1304:
1305:     memcpy(ptr, Z_STRVAL(prefix), Z_STRLEN(prefix));
1306:     ptr += Z_STRLEN(prefix);
1307:     memcpy(ptr, Z_STRVAL(key), Z_STRLEN(key));
1308:     ptr += Z_STRLEN(key);
1309:     memcpy(ptr, Z_STRVAL(postfix), Z_STRLEN(postfix));
1310:     ptr += Z_STRLEN(postfix);
1311:     *ptr = 0;
1312:
1313:     zval_ptr_dtor(&prefix);
1314:     zval_ptr_dtor(&key);
1315:     zval_ptr_dtor(&postfix);
1316:
1317:     RETURN_NEW_STR(str);
1318:     /* '}' */
1319:
1320:     /* {{{ proto void RecursiveTreeIterator::rewind()
1321:  Rewind the iterator to the first element */
1322: SPL_METHOD (RecursiveTreeIterator, rewind)
1323: {
1324:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1325:
1326:     if (zend_parse_parameters_none() == FAILURE) {
1327:         return;
1328:     }
1329:
1330:     if(object->it-iterators) {
1331:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1332:             "The object is in an invalid state as the parent constructor was not called");
1333:         return;
1334:     }
1335:
1336:     spl_recursive_tree_iterator_rewind(object);
1337:     /* '}' */
1338:
1339:     /* {{{ proto void RecursiveTreeIterator::valid()
1340:  Checks if the iterator is valid (not exhausted) */
1341: SPL_METHOD (RecursiveTreeIterator, valid)
1342: {
1343:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1344:
1345:     if (zend_parse_parameters_none() == FAILURE) {
1346:         return;
1347:     }
1348:
1349:     if(object->it-iterators) {
1350:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1351:             "The object is in an invalid state as the parent constructor was not called");
1352:         return;
1353:     }
1354:
1355:     spl_recursive_tree_iterator_valid(object);
1356:     /* '}' */
1357:
1358:     /* {{{ proto void RecursiveTreeIterator::current()
1359:  Returns the current element */
1360: SPL_METHOD (RecursiveTreeIterator, current)
1361: {
1362:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1363:
1364:     if (zend_parse_parameters_none() == FAILURE) {
1365:         return;
1366:     }
1367:
1368:     if(object->it-iterators) {
1369:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1370:             "The object is in an invalid state as the parent constructor was not called");
1371:         return;
1372:     }
1373:
1374:     spl_recursive_tree_iterator_current(object);
1375:     /* '}' */
1376:
1377:     /* {{{ proto void RecursiveTreeIterator::next()
1378:  Move the iterator to the next element */
1379: SPL_METHOD (RecursiveTreeIterator, next)
1380: {
1381:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1382:
1383:     if (zend_parse_parameters_none() == FAILURE) {
1384:         return;
1385:     }
1386:
1387:     if(object->it-iterators) {
1388:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1389:             "The object is in an invalid state as the parent constructor was not called");
1390:         return;
1391:     }
1392:
1393:     spl_recursive_tree_iterator_next(object);
1394:     /* '}' */
1395:
1396:     /* {{{ proto void RecursiveTreeIterator::previous()
1397:  Move the iterator to the previous element */
1398: SPL_METHOD (RecursiveTreeIterator, previous)
1399: {
1400:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1401:
1402:     if (zend_parse_parameters_none() == FAILURE) {
1403:         return;
1404:     }
1405:
1406:     if(object->it-iterators) {
1407:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1408:             "The object is in an invalid state as the parent constructor was not called");
1409:         return;
1410:     }
1411:
1412:     spl_recursive_tree_iterator_previous(object);
1413:     /* '}' */
1414:
1415:     /* {{{ proto void RecursiveTreeIterator::key()
1416:  Returns the current key */
1417: SPL_METHOD (RecursiveTreeIterator, key)
1418: {
1419:     spl_recursive_it_object *object = Z_SPLRECURSIVE_IT_P(getThis());
1420:
1421:     if (zend_parse_parameters_none() == FAILURE) {
1422:         return;
1423:     }
1424:
1425:     if(object->it-iterators) {
1426:         zend_throw_exception_ex(spl_ce_LogicException, 0,
1427:             "The object is in an invalid state as the
```

```

1489: zend_throw_exception(spl_ce_invalidArgumentException, "Flags must contain only one of CALL_TO_STRING, TO_STRING_USE_KEY, TO_STRING_USE_CURRENT, TO
1490:   STRING_USE_INDEX", 0);
1491: return NULL;
1492: }
1493: }
1494: }
1495: }
1496: }
1497: }
1498: }
1499: }
1500: }
1501: }
1502: }
1503: }
1504: }
1505: }
1506: }
1507: }
1508: }
1509: }
1510: }
1511: }
1512: }
1513: }
1514: }
1515: }
1516: }
1517: }
1518: }
1519: }
1520: }
1521: }
1522: }
1523: }
1524: }
1525: }
1526: }
1527: }
1528: }
1529: }
1530: }
1531: }
1532: }
1533: }
1534: }
1535: }
1536: }
1537: }
1538: }
1539: }
1540: }
1541: }
1542: }
1543: }
1544: }
1545: }
1546: }
1547: }
1548: }
1549: }
1550: }
1551: }
1552: }
1553: }
1554: }
1555: }
1556: }
1557: }
1558: }
1559: }
1560: }
1561: }
1562: }
1563: }
1564: }
1565: }
1566: }
1567: }
1568: }
1569: }
1570: }
1571: }
1572: }
1573: }
1574: }
1575: }
1576: }
1577: }
1578: }
1579: }
1580: }
1581: }
1582: }
1583: }
1584: }
1585: }
1586: }
1587: }
1588: }
1589: }
1590: }
1591: }
1592: }
1593: }
1594: }
1595: }
1596: }
1597: }
1598: }
1599: }
1600: }
1601: }
1602: }
1603: }
1604: }
1605: }
1606: }
1607: }
1608: }
1609: }
1610: }
1611: }
1612: }
1613: }
1614: }
1615: }
1616: }
1617: }
1618: }
1619: }
1620: }
1621: }
1622: }
1623: }
1624: }
1625: }
1626: }
1627: }
1628: }
1629: }
1630: }
1631: }
1632: }
1633: }
1634: }
1635: }
1636: }
1637: }
1638: }
1639: }
1640: }
1641: }
1642: }
1643: }
1644: }
1645: }
1646: }
1647: }
1648: }
1649: }
1650: }
1651: }
1652: }
1653: }
1654: }
1655: }
1656: }
1657: }
1658: }
1659: }
1660: }
1661: }
1662: }
1663: }
1664: }
1665: }
1666: }
1667: }
1668: }
1669: }
1670: }
1671: }
1672: }
1673: }
1674: }
1675: }
1676: }
1677: }
1678: }
1679: }
1680: }
1681: }
1682: }
1683: }
1684: }
1685: }
1686: }
1687: }
1688: }
1689: }
1690: }
1691: }
1692: }
1693: }
1694: }
1695: }
1696: }
1697: }
1698: }
1699: }
1700: }
1701: }
1702: }
1703: }
1704: }
1705: }
1706: }
1707: }
1708: }
1709: }
1710: }
1711: }
1712: }
1713: }
1714: }
1715: }
1716: }
1717: }
1718: }
1719: }
1720: }
1721: }
1722: }
1723: }
1724: }
1725: }
1726: }
1727: }
1728: }
1729: }
1730: }
1731: }
1732: }
1733: }
1734: }
1735: }
1736: }
1737: }
1738: }
1739: }
1740: }
1741: }
1742: }
1743: }
1744: }
1745: }
1746: }
1747: }
1748: }
1749: }
1750: }
1751: }
1752: }
1753: }
1754: }
1755: }
1756: }
1757: }
1758: }
1759: }
1760: }
1761: }
1762: }
1763: }
1764: }
1765: }
1766: }
1767: }
1768: }
1769: }
1770: }
1771: }
1772: }
1773: }
1774: }
1775: }
1776: }
1777: }
1778: }
1779: }
1780: }
1781: }
1782: }
1783: }
1784: }
1785: }
1786: }
1787: }
1788: }
1789: }
1790: }
1791: }
1792: }
1793: }
1794: }
1795: }
1796: }
1797: }
1798: }
1799: }
1800: }
1801: }
1802: }
1803: }
1804: }
1805: }
1806: }
1807: }
1808: }
1809: }
1810: }
1811: }
1812: }
1813: }
1814: }
1815: }
1816: }
1817: }
1818: }
1819: }
1820: }
1821: }
1822: }
1823: }
1824: }
1825: }
1826: }
1827: }
1828: }
1829: }
1830: }
1831: }
1832: }
1833: }
1834: }
1835: }
1836: }
1837: }
1838: }
1839: }
1840: }
1841: }
1842: }
1843: }
1844: }
1845: }
1846: }
1847: }
1848: }
1849: }
1850: }
1851: }
1852: }
1853: }
1854: }
1855: }
1856: }
1857: }
1858: }
1859: }
1860: }
1861: }
1862: }
1863: }
1864: }
1865: }
1866: }
1867: }
1868: }
1869: }
1870: }

```

```

1683: }
1684: }
1685: }
1686: }
1687: }
1688: }
1689: }
1690: }
1691: }
1692: }
1693: }
1694: }
1695: }
1696: }
1697: }
1698: }
1699: }
1700: }
1701: }
1702: }
1703: }
1704: }
1705: }
1706: }
1707: }
1708: }
1709: }
1710: }
1711: }
1712: }
1713: }
1714: }
1715: }
1716: }
1717: }
1718: }
1719: }
1720: }
1721: }
1722: }
1723: }
1724: }
1725: }
1726: }
1727: }
1728: }
1729: }
1730: }
1731: }
1732: }
1733: }
1734: }
1735: }
1736: }
1737: }
1738: }
1739: }
1740: }
1741: }
1742: }
1743: }
1744: }
1745: }
1746: }
1747: }
1748: }
1749: }
1750: }
1751: }
1752: }
1753: }
1754: }
1755: }
1756: }
1757: }
1758: }
1759: }
1760: }
1761: }
1762: }
1763: }
1764: }
1765: }
1766: }
1767: }
1768: }
1769: }
1770: }
1771: }
1772: }
1773: }
1774: }
1775: }
1776: }
1777: }
1778: }
1779: }
1780: }
1781: }
1782: }
1783: }
1784: }
1785: }
1786: }
1787: }
1788: }
1789: }
1790: }
1791: }
1792: }
1793: }
1794: }
1795: }
1796: }
1797: }
1798: }
1799: }
1800: }
1801: }
1802: }
1803: }
1804: }
1805: }
1806: }
1807: }
1808: }
1809: }
1810: }
1811: }
1812: }
1813: }
1814: }
1815: }
1816: }
1817: }
1818: }
1819: }
1820: }
1821: }
1822: }
1823: }
1824: }
1825: }
1826: }
1827: }
1828: }
1829: }
1830: }
1831: }
1832: }
1833: }
1834: }
1835: }
1836: }
1837: }
1838: }
1839: }
1840: }
1841: }
1842: }
1843: }
1844: }
1845: }
1846: }
1847: }
1848: }
1849: }
1850: }
1851: }
1852: }
1853: }
1854: }
1855: }
1856: }
1857: }
1858: }
1859: }
1860: }
1861: }
1862: }
1863: }
1864: }
1865: }
1866: }
1867: }
1868: }
1869: }
1870: }

```

```

1871:     spl_filter_it_fetch(this, intern);
1872: }
1873:
1874: /* {{{ proto void FilterIterator::rewind()
1875:  * Rewind the iterator */
1876: SPL_METHOD(FilterIterator, rewind)
1877: {
1878:     spl_dual_it_object *intern;
1879:
1880:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
1881:         return;
1882:     }
1883:
1884:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1885:     spl_filter_it_rewind(getThis(), intern);
1886:     /* }}} */
1887:
1888: /* {{{ proto void FilterIterator::next()
1889:  * Move the iterator forward */
1890: SPL_METHOD(FilterIterator, next)
1891: {
1892:     spl_dual_it_object *intern;
1893:
1894:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
1895:         return;
1896:     }
1897:
1898:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1899:     spl_filter_it_next(getThis(), intern);
1900:     /* }}} */
1901:
1902: /* {{{ proto void RecursiveCallbackFilterIterator::__construct(RecursiveIterator it, callback func)
1903:  * Create a RecursiveCallbackFilterIterator from a RecursiveIterator */
1904: SPL_METHOD(RecursiveCallbackFilterIterator, __construct)
1905: {
1906:     spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveCallbackFilterIterator, spl_ca_RecursiveIterator, DIT_RecursiveCallbackFilter
1907: ) /* }}} */
1908:
1909:
1910: /* {{{ proto void RecursiveFilterIterator::__construct(RecursiveIterator it)
1911:  * Create a RecursiveFilterIterator from a RecursiveIterator */
1912: SPL_METHOD(RecursiveFilterIterator, __construct)
1913: {
1914:     spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveFilterIterator, spl_ca_RecursiveIterator, DIT_RecursiveFilterIterator);
1915:     /* }}} */
1916:
1917: /* {{{ proto bool RecursiveFilterIterator::hasChildren()
1918:  * Check whether the inner iterator's current element has children */
1919: SPL_METHOD(RecursiveFilterIterator, hasChildren)
1920: {
1921:     spl_dual_it_object *intern;
1922:     zval retval;
1923:
1924:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
1925:         return;
1926:     }
1927:
1928:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1929:
1930:     zend_call_method_with_0_params(&intern->inner.object, intern->inner.ce, NULL, "haschildren", &retval);
1931:     IF (Z_TYPE(retval) != IS_UNDEF) {
1932:         RETURN_ZVAL(&retval, 0, 1);
1933:     } else {
1934:         RETURN_FALSE;
1935:     }
1936:     /* }}} */
1937:
1938: /* {{{ proto RecursiveFilterIterator RecursiveFilterIterator::getChildren()
1939:  * Return the inner iterator's children contained in a RecursiveFilterIterator */
1940: SPL_METHOD(RecursiveFilterIterator, getChildren)
1941: {
1942:     spl_dual_it_object *intern;
1943:     zval retval;
1944:
1945:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
1946:         return;
1947:     }
1948:
1949:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1950:
1951:     zend_call_method_with_0_params(&intern->inner.object, intern->inner.ce, NULL, "getchildren", &retval);
1952:     IF (EG(exception) && Z_TYPE(retval) != IS_UNDEF) {
1953:         spl_instantiate_arg_ext(Z_OBJCE_P(getThis()), return_value, &retval);
1954:     }
1955:     zend_get_zfor(&retval);
1956:     /* }}} */
1957:
1958: /* {{{ proto RecursiveCallbackFilterIterator RecursiveCallbackFilterIterator::getChildren()
1959:  * Return the inner iterator's children contained in a RecursiveCallbackFilterIterator */
1960: SPL_METHOD(RecursiveCallbackFilterIterator, getChildren)
1961: {
1962:     spl_dual_it_object *intern;
1963:     zval retval;
1964:
1965:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
1966:         return;
1967:     }
1968:
1969:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
1970:
1971:     zend_call_method_with_0_params(&intern->inner.object, intern->inner.ce, NULL, "getchildren", &retval);
1972:     IF (EG(exception) && Z_TYPE(retval) != IS_UNDEF) {
1973:         spl_instantiate_arg_ext(Z_OBJCE_P(getThis()), return_value, &retval, &intern->callback->fci.function_name);
1974:     }
1975:     zend_get_zfor(&retval);
1976:     /* }}} */
1977:
1978: /* {{{ proto void ParentIterator::__construct(RecursiveIterator it)
1979:  * Create a ParentIterator from a RecursiveIterator */
1980: SPL_METHOD(ParentIterator, __construct)
1981: {
1982:     spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_ParentIterator, spl_ca_RecursiveIterator, DIT_ParentIterator);
1983:     /* }}} */
1984:
1985: /* {{{ proto void RegexIterator::__construct(Iterator it, string regex [, int mode [, int flags [, int preg_flags]]])
1986:  * Create an RegexIterator from another iterator and a regular expression */
1987: SPL_METHOD(RegexIterator, __construct)
1988: {
1989:     spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RegexIterator, zend_ca_iterator, DIT_RegexIterator);
1990:     /* }}} */
1991:
1992: /* {{{ proto bool CallbackFilterIterator::accept()
1993:  * Call the callback with the current value, the current key and the inner iterator as arguments */
1994: SPL_METHOD(CallbackFilterIterator, accept)
1995: {
1996:     spl_dual_it_object *intern = Z_SPDUAL_IT_P(getThis());
1997:     zend_fcall_info *fci = &intern->callback->fci;
1998:     zend_fcall_info_cache *foc = &intern->callback->foc;
1999:     zval params[3];
2000:
2001:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
2002:         return;
2003:     }
2004:
2005:     IF (Z_TYPE(intern->current.data) == IS_UNDEF || Z_TYPE(intern->current.key) == IS_UNDEF) {
2006:         RETURN_FALSE;
2007:     }
2008:
2009:     ZVAL_COPY_VALUE(&params[0], &intern->current.data);
2010:     ZVAL_COPY_VALUE(&params[1], &intern->current.key);
2011:     ZVAL_COPY_VALUE(&params[2], &intern->inner.object);
2012:
2013:     fci->retval = return_value;
2014:     fci->param_count = 3;
2015:     fci->params = params;
2016:     fci->no_separation = 0;
2017:
2018:     IF (zend_call_function(fci, foc) != SUCCESS || Z_ISUNDEF(return_value)) {
2019:         RETURN_FALSE;
2020:     }
2021:
2022:     IF (EG(exception)) {
2023:         RETURN_NULL();
2024:     }
2025:
2026:     /* zend_call_function may change args to IS_REF */
2027:     ZVAL_COPY_VALUE(&intern->current.data, &params[0]);
2028:     ZVAL_COPY_VALUE(&intern->current.key, &params[1]);
2029:     /* }}} */
2030:
2031:
2032: /* {{{ proto bool RegexIterator::accept()
2033:  * Match (string)current() against regular expression */
2034: SPL_METHOD(RegexIterator, accept)
2035: {
2036:     spl_dual_it_object *intern;
2037:     zend_string *result, *subject;
2038:     zval count = 0;
2039:     zval account, *replacement, tmp_replacement, rv;
2040:     pcre_match_data *match_data;
2041:     pcre_code *re;
2042:     int rc;
2043:
2044:     IF_SEND_PARAM_PARAMETERS_NONE() == FAILURE() {
2045:         return;
2046:     }
2047:
2048:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2049:
2050:     IF (Z_TYPE(intern->current.data) == IS_UNDEF) {
2051:         RETURN_FALSE;
2052:     }
2053:
2054:     IF (intern->u.regex.flags & REGIT_USE_KEY) {
2055:         subject = zval_get_string(&intern->current.key);
2056:     } else {
2057:         IF (Z_TYPE(intern->current.data) == IS_ARRAY) {
2058:             RETURN_FALSE;
2059:         }
2060:
2061:         RETURN_FALSE;
2062:     }
2063:
2064:     switch (intern->u.regex.mode) {
2065:         case REGIT_MODE_MAX: /* won't happen but makes compiler happy */
2066:         case REGIT_MODE_MATCH:
2067:             re = pcre_get_pcre_re(intern->u.regex.pcre);
2068:             match_data = pcre_create_match_data(0, re);
2069:             IF (!match_data) {
2070:                 RETURN_FALSE;
2071:             }
2072:             rc = pcre2_match(re, (PCRE2_SPTR)ZSTR_VAL(subject), ZSTR_LEN(subject), 0, 0, match_data, pcre_pcre_match());
2073:             RETVAL_BOOL(rc == 0);
2074:             pcre_free(match_data);
2075:             break;
2076:         case REGIT_MODE_ALL_MATCHES:
2077:         case REGIT_MODE_GET_MATCH:
2078:             ZVAL_UNDEF(&intern->current.data);
2079:             ZVAL_UNDEF(&intern->current.data);
2080:             pcre_pcre_match_impl(intern->u.regex.pcre, ZSTR_VAL(subject), ZSTR_LEN(subject), &account,
2081:                 &intern->current.data, &intern->u.regex.mode == REGIT_MODE_ALL_MATCHES, intern->u.regex.use_flags, intern->u.regex.preg_flags, 0);
2082:             RETVAL_BOOL(&account > 0);
2083:             break;
2084:         case REGIT_MODE_SPLIT:
2085:             zval_get_zfor(&intern->current.data);
2086:             ZVAL_UNDEF(&intern->current.data);
2087:             pcre_pcre_split_impl(intern->u.regex.pcre, subject, &intern->current.data, -1, intern->u.regex.preg_flags);
2088:             count = zend_hash_num_elements(Z_ARRVAL(intern->current.data));
2089:             RETVAL_BOOL(count > 1);
2090:             break;
2091:         case REGIT_MODE_REPLACE:
2092:             replacement = zend_read_property(intern->std.ce, getThis(), "replacement", sizeof("replacement")-1, 1, &rv);
2093:             IF (Z_TYPE_P(replacement) != IS_STRING) {
2094:                 ZVAL_COPY(&tmp_replacement, replacement);
2095:                 convert_to_string(&tmp_replacement);
2096:                 replacement = &tmp_replacement;
2097:             }
2098:             result = pcre_pcre_replace_impl(intern->u.regex.pcre, subject, ZSTR_VAL(subject), ZSTR_LEN(subject), ZSTR_P(replacement), -1, &account);
2099:             IF (intern->u.regex.flags & REGIT_USE_KEY) {
2100:                 zval_get_zfor(&intern->current.key);
2101:                 ZVAL_STR(&intern->current.key, result);
2102:             } else {
2103:                 zval_get_zfor(&intern->current.data);
2104:                 ZVAL_STR(&intern->current.data, result);
2105:             }
2106:             IF (replacement == &tmp_replacement) {
2107:                 zval_get_zfor(&replacement);
2108:             }
2109:             RETVAL_BOOL(count > 0);
2110:             break;
2111:         case REGIT_MODE_INVERTED:
2112:             RETVAL_BOOL(Z_TYPE_P(return_value) != IS_TRUE);
2113:             zend_string_release(subject);
2114:             /* }}} */
2115:         case REGIT_MODE_REPLACE:
2116:             /* {{{ proto string RegexIterator::getRegex()
2117:              * Returns current regular expression */
2118:             SPL_METHOD(RegexIterator, getRegex)
2119:             {
2120:                 spl_dual_it_object *intern = Z_SPDUAL_IT_P(getThis());
2121:                 IF (zend_param_parameters_none() == FAILURE) {
2122:                     return;
2123:                 }
2124:                 RETURN_STR_COPY(&intern->u.regex.regex);
2125:             } /* }}} */
2126:
2127:         /* {{{ proto bool RegexIterator::getMode()
2128:          * Returns current operation mode */
2129:             SPL_METHOD(RegexIterator, getMode)
2130:             {
2129:                 spl_dual_it_object *intern;
2130:
2131:                 IF (zend_param_parameters_none() == FAILURE) {
2132:                     return;
2133:                 }
2134:                 RETURN_LONG(intern->u.regex.mode);
2135:             } /* }}} */
2136:
2137:         /* {{{ proto bool RegexIterator::setMode(int new_mode)
2138:          * Set new operation mode */
2139:             SPL_METHOD(RegexIterator, setMode)
2140:             {
2141:                 spl_dual_it_object *intern;
2142:                 zend_long mode;
2143:
2144:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &mode) == FAILURE) {
2145:                     return;
2146:                 }
2147:                 IF (mode < 0 || mode >= REGIT_MODE_MAX) {
2148:                     zend_throw_exception_ext(spl_ca_invalidArgumentException, 0, "Illegal mode " ZEND_LONG_FMT, mode);
2149:                     return; /* NULL */
2150:                 }
2151:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2152:                 intern->u.regex.mode = mode;
2153:             } /* }}} */
2154:
2155:         /* {{{ proto bool RegexIterator::getFlags()
2156:          * Returns current operation flags */
2157:             SPL_METHOD(RegexIterator, getFlags)
2158:             {
2159:                 spl_dual_it_object *intern;
2160:
2161:                 IF (zend_param_parameters_none() == FAILURE) {
2162:                     return;
2163:                 }
2164:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2165:                 RETURN_LONG(intern->u.regex.flags);
2166:             } /* }}} */
2167:
2168:         /* {{{ proto bool RegexIterator::setFlags(int new_flags)
2169:          * Set operation flags */
2170:             SPL_METHOD(RegexIterator, setFlags)
2171:             {
2172:                 spl_dual_it_object *intern;
2173:                 zend_long flags;
2174:
2175:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &flags) == FAILURE) {
2176:                     return;
2177:                 }
2178:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2179:                 intern->u.regex.flags = flags;
2180:             } /* }}} */
2181:
2182:         /* {{{ proto bool RegexIterator::getPregFlags()
2183:          * Returns current PREG flags (if in use or NULL) */
2184:             SPL_METHOD(RegexIterator, getPregFlags)
2185:             {
2186:                 spl_dual_it_object *intern;
2187:
2188:                 IF (zend_param_parameters_none() == FAILURE) {
2189:                     return;
2190:                 }
2191:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2192:                 RETURN_LONG(intern->u.regex.preg_flags);
2193:             } /* }}} */
2194:
2195:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2196:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2197:             SPL_METHOD(RecursiveRegexIterator, __construct)
2198:             {
2199:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2200:                 /* }}} */
2201:             }
2202:
2203:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2204:          * Set PREG flags */
2205:             SPL_METHOD(RegexIterator, setPregFlags)
2206:             {
2207:                 spl_dual_it_object *intern;
2208:
2209:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2210:                     return;
2211:                 }
2212:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2213:                 intern->u.regex.preg_flags = preg_flags;
2214:                 intern->u.regex.use_flags = 1;
2215:             } /* }}} */
2216:
2217:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2218:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2219:             SPL_METHOD(RecursiveRegexIterator, __construct)
2220:             {
2221:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2222:                 /* }}} */
2223:             }
2224:
2225:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2226:          * Set PREG flags */
2227:             SPL_METHOD(RegexIterator, setPregFlags)
2228:             {
2229:                 spl_dual_it_object *intern;
2230:                 zend_long preg_flags;
2231:
2232:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2233:                     return;
2234:                 }
2235:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2236:                 intern->u.regex.preg_flags = preg_flags;
2237:                 intern->u.regex.use_flags = 1;
2238:             } /* }}} */
2239:
2240:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2241:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2242:             SPL_METHOD(RecursiveRegexIterator, __construct)
2243:             {
2244:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2245:                 /* }}} */
2246:             }
2247:
2248:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2249:          * Set PREG flags */
2250:             SPL_METHOD(RegexIterator, setPregFlags)
2251:             {
2252:                 spl_dual_it_object *intern;
2253:                 zend_long preg_flags;
2254:
2255:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2256:                     return;
2257:                 }
2258:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2259:                 intern->u.regex.preg_flags = preg_flags;
2260:                 intern->u.regex.use_flags = 1;
2261:             } /* }}} */
2262:
2263:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2264:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2265:             SPL_METHOD(RecursiveRegexIterator, __construct)
2266:             {
2267:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2268:                 /* }}} */
2269:             }
2270:
2271:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2272:          * Set PREG flags */
2273:             SPL_METHOD(RegexIterator, setPregFlags)
2274:             {
2275:                 spl_dual_it_object *intern;
2276:                 zend_long preg_flags;
2277:
2278:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2279:                     return;
2280:                 }
2281:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2282:                 intern->u.regex.preg_flags = preg_flags;
2283:                 intern->u.regex.use_flags = 1;
2284:             } /* }}} */
2285:
2286:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2287:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2288:             SPL_METHOD(RecursiveRegexIterator, __construct)
2289:             {
2290:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2291:                 /* }}} */
2292:             }
2293:
2294:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2295:          * Set PREG flags */
2296:             SPL_METHOD(RegexIterator, setPregFlags)
2297:             {
2298:                 spl_dual_it_object *intern;
2299:                 zend_long preg_flags;
2300:
2301:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2302:                     return;
2303:                 }
2304:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2305:                 intern->u.regex.preg_flags = preg_flags;
2306:                 intern->u.regex.use_flags = 1;
2307:             } /* }}} */
2308:
2309:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2310:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2311:             SPL_METHOD(RecursiveRegexIterator, __construct)
2312:             {
2313:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2314:                 /* }}} */
2315:             }
2316:
2317:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2318:          * Set PREG flags */
2319:             SPL_METHOD(RegexIterator, setPregFlags)
2320:             {
2321:                 spl_dual_it_object *intern;
2322:                 zend_long preg_flags;
2323:
2324:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2325:                     return;
2326:                 }
2327:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2328:                 intern->u.regex.preg_flags = preg_flags;
2329:                 intern->u.regex.use_flags = 1;
2330:             } /* }}} */
2331:
2332:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2333:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2334:             SPL_METHOD(RecursiveRegexIterator, __construct)
2335:             {
2336:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2337:                 /* }}} */
2338:             }
2339:
2340:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2341:          * Set PREG flags */
2342:             SPL_METHOD(RegexIterator, setPregFlags)
2343:             {
2344:                 spl_dual_it_object *intern;
2345:                 zend_long preg_flags;
2346:
2347:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2348:                     return;
2349:                 }
2350:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2351:                 intern->u.regex.preg_flags = preg_flags;
2352:                 intern->u.regex.use_flags = 1;
2353:             } /* }}} */
2354:
2355:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2356:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2357:             SPL_METHOD(RecursiveRegexIterator, __construct)
2358:             {
2359:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2360:                 /* }}} */
2361:             }
2362:
2363:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2364:          * Set PREG flags */
2365:             SPL_METHOD(RegexIterator, setPregFlags)
2366:             {
2367:                 spl_dual_it_object *intern;
2368:                 zend_long preg_flags;
2369:
2370:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2371:                     return;
2372:                 }
2373:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2374:                 intern->u.regex.preg_flags = preg_flags;
2375:                 intern->u.regex.use_flags = 1;
2376:             } /* }}} */
2377:
2378:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2379:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2380:             SPL_METHOD(RecursiveRegexIterator, __construct)
2381:             {
2382:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2383:                 /* }}} */
2384:             }
2385:
2386:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2387:          * Set PREG flags */
2388:             SPL_METHOD(RegexIterator, setPregFlags)
2389:             {
2390:                 spl_dual_it_object *intern;
2391:                 zend_long preg_flags;
2392:
2393:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2394:                     return;
2395:                 }
2396:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2397:                 intern->u.regex.preg_flags = preg_flags;
2398:                 intern->u.regex.use_flags = 1;
2399:             } /* }}} */
2400:
2401:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2402:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2403:             SPL_METHOD(RecursiveRegexIterator, __construct)
2404:             {
2405:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2406:                 /* }}} */
2407:             }
2408:
2409:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2410:          * Set PREG flags */
2411:             SPL_METHOD(RegexIterator, setPregFlags)
2412:             {
2413:                 spl_dual_it_object *intern;
2414:                 zend_long preg_flags;
2415:
2416:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2417:                     return;
2418:                 }
2419:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2420:                 intern->u.regex.preg_flags = preg_flags;
2421:                 intern->u.regex.use_flags = 1;
2422:             } /* }}} */
2423:
2424:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2425:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2426:             SPL_METHOD(RecursiveRegexIterator, __construct)
2427:             {
2428:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2429:                 /* }}} */
2430:             }
2431:
2432:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2433:          * Set PREG flags */
2434:             SPL_METHOD(RegexIterator, setPregFlags)
2435:             {
2436:                 spl_dual_it_object *intern;
2437:                 zend_long preg_flags;
2438:
2439:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2440:                     return;
2441:                 }
2442:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2443:                 intern->u.regex.preg_flags = preg_flags;
2444:                 intern->u.regex.use_flags = 1;
2445:             } /* }}} */
2446:
2447:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2448:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2449:             SPL_METHOD(RecursiveRegexIterator, __construct)
2450:             {
2451:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2452:                 /* }}} */
2453:             }
2454:
2455:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2456:          * Set PREG flags */
2457:             SPL_METHOD(RegexIterator, setPregFlags)
2458:             {
2459:                 spl_dual_it_object *intern;
2460:                 zend_long preg_flags;
2461:
2462:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2463:                     return;
2464:                 }
2465:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2466:                 intern->u.regex.preg_flags = preg_flags;
2467:                 intern->u.regex.use_flags = 1;
2468:             } /* }}} */
2469:
2470:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2471:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2472:             SPL_METHOD(RecursiveRegexIterator, __construct)
2473:             {
2474:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2475:                 /* }}} */
2476:             }
2477:
2478:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2479:          * Set PREG flags */
2480:             SPL_METHOD(RegexIterator, setPregFlags)
2481:             {
2482:                 spl_dual_it_object *intern;
2483:                 zend_long preg_flags;
2484:
2485:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2486:                     return;
2487:                 }
2488:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2489:                 intern->u.regex.preg_flags = preg_flags;
2490:                 intern->u.regex.use_flags = 1;
2491:             } /* }}} */
2492:
2493:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2494:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2495:             SPL_METHOD(RecursiveRegexIterator, __construct)
2496:             {
2497:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2498:                 /* }}} */
2499:             }
2500:
2501:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2502:          * Set PREG flags */
2503:             SPL_METHOD(RegexIterator, setPregFlags)
2504:             {
2505:                 spl_dual_it_object *intern;
2506:                 zend_long preg_flags;
2507:
2508:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2509:                     return;
2510:                 }
2511:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2512:                 intern->u.regex.preg_flags = preg_flags;
2513:                 intern->u.regex.use_flags = 1;
2514:             } /* }}} */
2515:
2516:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2517:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2518:             SPL_METHOD(RecursiveRegexIterator, __construct)
2519:             {
2520:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2521:                 /* }}} */
2522:             }
2523:
2524:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2525:          * Set PREG flags */
2526:             SPL_METHOD(RegexIterator, setPregFlags)
2527:             {
2528:                 spl_dual_it_object *intern;
2529:                 zend_long preg_flags;
2530:
2531:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2532:                     return;
2533:                 }
2534:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2535:                 intern->u.regex.preg_flags = preg_flags;
2536:                 intern->u.regex.use_flags = 1;
2537:             } /* }}} */
2538:
2539:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2540:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2541:             SPL_METHOD(RecursiveRegexIterator, __construct)
2542:             {
2543:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2544:                 /* }}} */
2545:             }
2546:
2547:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2548:          * Set PREG flags */
2549:             SPL_METHOD(RegexIterator, setPregFlags)
2550:             {
2551:                 spl_dual_it_object *intern;
2552:                 zend_long preg_flags;
2553:
2554:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2555:                     return;
2556:                 }
2557:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2558:                 intern->u.regex.preg_flags = preg_flags;
2559:                 intern->u.regex.use_flags = 1;
2560:             } /* }}} */
2561:
2562:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2563:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2564:             SPL_METHOD(RecursiveRegexIterator, __construct)
2565:             {
2566:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2567:                 /* }}} */
2568:             }
2569:
2570:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2571:          * Set PREG flags */
2572:             SPL_METHOD(RegexIterator, setPregFlags)
2573:             {
2574:                 spl_dual_it_object *intern;
2575:                 zend_long preg_flags;
2576:
2577:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2578:                     return;
2579:                 }
2580:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2581:                 intern->u.regex.preg_flags = preg_flags;
2582:                 intern->u.regex.use_flags = 1;
2583:             } /* }}} */
2584:
2585:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2586:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2587:             SPL_METHOD(RecursiveRegexIterator, __construct)
2588:             {
2589:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2590:                 /* }}} */
2591:             }
2592:
2593:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2594:          * Set PREG flags */
2595:             SPL_METHOD(RegexIterator, setPregFlags)
2596:             {
2597:                 spl_dual_it_object *intern;
2598:                 zend_long preg_flags;
2599:
2600:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2601:                     return;
2602:                 }
2603:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2604:                 intern->u.regex.preg_flags = preg_flags;
2605:                 intern->u.regex.use_flags = 1;
2606:             } /* }}} */
2607:
2608:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2609:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2610:             SPL_METHOD(RecursiveRegexIterator, __construct)
2611:             {
2612:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2613:                 /* }}} */
2614:             }
2615:
2616:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2617:          * Set PREG flags */
2618:             SPL_METHOD(RegexIterator, setPregFlags)
2619:             {
2620:                 spl_dual_it_object *intern;
2621:                 zend_long preg_flags;
2622:
2623:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2624:                     return;
2625:                 }
2626:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2627:                 intern->u.regex.preg_flags = preg_flags;
2628:                 intern->u.regex.use_flags = 1;
2629:             } /* }}} */
2630:
2631:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2632:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2633:             SPL_METHOD(RecursiveRegexIterator, __construct)
2634:             {
2635:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2636:                 /* }}} */
2637:             }
2638:
2639:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2640:          * Set PREG flags */
2641:             SPL_METHOD(RegexIterator, setPregFlags)
2642:             {
2643:                 spl_dual_it_object *intern;
2644:                 zend_long preg_flags;
2645:
2646:                 IF (zend_param_parameters(SEND_NUM_ARGS(), "1", &preg_flags) == FAILURE) {
2647:                     return;
2648:                 }
2649:                 SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2650:                 intern->u.regex.preg_flags = preg_flags;
2651:                 intern->u.regex.use_flags = 1;
2652:             } /* }}} */
2653:
2654:         /* {{{ proto void RecursiveRegexIterator::__construct(RecursiveIterator it, string regex [, int mode [, int flags [, int preg_flags]]])
2655:          * Create an RecursiveRegexIterator from another recursive iterator and a regular expression */
2656:             SPL_METHOD(RecursiveRegexIterator, __construct)
2657:             {
2658:                 spl_dual_it_construct(INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ca_RecursiveRegexIterator, spl_ca_RecursiveIterator, DIT_RecursiveRegexIterator);
2659:                 /* }}} */
2660:             }
2661:
2662:         /* {{{ proto bool RegexIterator::setPregFlags(int new_flags)
2
```



```

2246: /* {{{ proto RecursiveHashIterator RecursiveHashIterator::__construct()
2247:  * Return the inner iterator's children contained in a RecursiveHashIterator */
2248: SPL_METHOD(RecursiveHashIterator, getChildren)
2249: {
2250:     spl_dual_it_object *intern;
2251:     zval rretval;
2252:
2253:     if (zend_parse_parameters_none() == FAILURE) {
2254:         return;
2255:     }
2256:
2257:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2258:
2259:     zend_call_method_with_0_params(intern->inner->obj, intern->inner->ce, NULL, "getChildren", &rretval);
2260:     if (!EG(exception)) {
2261:         zval args[5];
2262:
2263:         ZVAL_COPY(&args[0], &rretval);
2264:         ZVAL_STR_COPY(&args[1], intern->u.regex.regex);
2265:         ZVAL_LONG(&args[2], intern->u.regex.mode);
2266:         ZVAL_LONG(&args[3], intern->u.regex.flags);
2267:         ZVAL_LONG(&args[4], intern->u.regex.preg_flags);
2268:
2269:         spl_instantiate_arg_n(2, OBJ_CEP, getThis(), return_value, 5, args);
2270:
2271:         zval_ptr_dtor(&args[0]);
2272:         zval_ptr_dtor(&args[1]);
2273:         zval_ptr_dtor(&args[2]);
2274:         zval_ptr_dtor(&args[3]);
2275:     } /* }}} */
2276:
2277: SPL_METHOD(RecursiveHashIterator, accept)
2278: {
2279:     spl_dual_it_object *intern;
2280:
2281:     if (zend_parse_parameters_none() == FAILURE) {
2282:         return;
2283:     }
2284:
2285:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2286:
2287:     if (!EG(exception)) {
2288:         RETURN_FALSE;
2289:     } else if (!EG(exception)) {
2290:         RETURN_BOOL(zend_hash_num_elements(2_ARRAYVAL(intern->current->data)) > 0);
2291:     }
2292:
2293:     zend_call_method_with_0_params(getThis(), spl_ce_RegeIterator, NULL, "accept", return_value);
2294:
2295:     return;
2296: }
2297:
2298: /* {{{ spl_dual_it_dtor */
2299: static void spl_dual_it_dtor(zend_object *object)
2300: {
2301:     spl_dual_it_object *object = spl_dual_it_from_obj(object);
2302:
2303:     /* call standard dtor */
2304:     zend_objects_destroy_object(object);
2305:
2306:     spl_dual_it_free(object);
2307:
2308:     if (object->inner->iterator) {
2309:         zend_iterator_dtor(object->inner->iterator);
2310:     }
2311: }
2312: /* }}} */
2313:
2314: /* {{{ spl_dual_it_free_storage */
2315: static void spl_dual_it_free_storage(zend_object *object)
2316: {
2317:     spl_dual_it_object *object = spl_dual_it_from_obj(object);
2318:
2319:     if (!IS_UNDEF(object->inner->obj)) {
2320:         zval_ptr_dtor(object->inner->obj);
2321:     }
2322:
2323:     if (object->dit_type == DIT_AppendIterator) {
2324:         zend_iterator_dtor(object->u.append->iterator);
2325:     }
2326:     if (!EG(exception)) {
2327:         zval_ptr_dtor(object->u.append->array);
2328:     }
2329: }
2330:
2331: if (object->dit_type == DIT_CachingIterator) {
2332:     zval_ptr_dtor(object->u.caching->cache);
2333: }
2334:
2335: if (HAVE_PCRE || HAVE_RUNDLED_PCRE)
2336:     if (object->dit_type == DIT_RegeIterator || object->dit_type == DIT_RecursiveRegeIterator) {
2337:         if (object->u.regex->pos) {
2338:             php_pcre_pos_decr(object->u.regex->pos);
2339:         }
2340:         if (object->u.regex->pos) {
2341:             zend_string_release(object->u.regex->regex);
2342:         }
2343:     }
2344: }
2345:
2346: if (object->dit_type == DIT_CallbackFilterIterator) {
2347:     if (object->u.cbfilter) {
2348:         zend_callbackfilter_dtor(intern->cbfilter->obj->u.cbfilter);
2349:         object->u.cbfilter = NULL;
2350:         zval_ptr_dtor(&cbfilter->obj->u.cbfilter->function_name);
2351:         if (cbfilter->obj->u.cbfilter) {
2352:             OBJ_RELEASE(cbfilter->obj->u.cbfilter);
2353:         }
2354:         zval_ptr_dtor(&cbfilter);
2355:     }
2356: }
2357:
2358: zend_object_std_dtor(object->std);
2359: }
2360: /* }}} */
2361:
2362: /* {{{ spl_dual_it_new */
2363: static zend_object *spl_dual_it_new(zend_class_entry *class_type)
2364: {
2365:     spl_dual_it_object *intern;
2366:
2367:     intern = zend_object_alloc(sizeof(spl_dual_it_object), class_type);
2368:     intern->dit_type = DIT_Unknown;
2369:
2370:     zend_object_std_init(intern->std, class_type);
2371:     object_properties_init(intern->std, class_type);
2372:
2373:     intern->std->handlers = spl_handlers_dual_it;
2374:     return intern->std;
2375: }
2376: /* }}} */
2377:
2378: ZEND_BEGIN_ARG_INFO(arginfo_filter_it__construct, 0)
2379:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
2380: ZEND_END_ARG_INFO();
2381:
2382: static const zend_function_entry spl_funcs_FilterIterator[] = {
2383:     SPL_ME(FilterIterator, __construct, arginfo_filter_it__construct, ZEND_ACC_PUBLIC)
2384:     SPL_ME(FilterIterator, rewind, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2385:     SPL_ME(dual_it, valid, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2386:     SPL_ME(dual_it, key, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2387:     SPL_ME(dual_it, current, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2388:     SPL_ME(FilterIterator, next, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2389:     SPL_ME(dual_it, getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2390:     SPL_ABSTRACT_ME(FilterIterator, accept, arginfo_recursive_it_void)
2391:     PHP_FE_END
2392: };
2393:
2394: ZEND_BEGIN_ARG_INFO(arginfo_callback_filter_it__construct, 0)
2395:     ZEND_ARG_OBJ_INFO(0, iterator, Iterator, 0)
2396:     ZEND_ARG_INFO(0, callback)
2397: ZEND_END_ARG_INFO();
2398:
2399: static const zend_function_entry spl_funcs_CallbackFilterIterator[] = {
2400:     SPL_ME(CallbackFilterIterator, __construct, arginfo_callback_filter_it__construct, ZEND_ACC_PUBLIC)
2401:     SPL_ME(CallbackFilterIterator, accept, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2402:     PHP_FE_END
2403: };
2404:
2405: ZEND_BEGIN_ARG_INFO(arginfo_recursive_callback_filter_it__construct, 0)
2406:     ZEND_ARG_OBJ_INFO(0, iterator, RecursiveIterator, 0)
2407:     ZEND_ARG_INFO(0, callback)
2408: ZEND_END_ARG_INFO();
2409:
2410: static const zend_function_entry spl_funcs_RecursiveCallbackFilterIterator[] = {
2411:     SPL_ME(RecursiveCallbackFilterIterator, __construct, arginfo_recursive_callback_filter_it__construct, ZEND_ACC_PUBLIC)
2412:     SPL_ME(RecursiveCallbackFilterIterator, hasChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2413:     SPL_ME(RecursiveCallbackFilterIterator, getChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2414:     PHP_FE_END
2415: };
2416:
2417: ZEND_BEGIN_ARG_INFO(arginfo_parent_it__construct, 0)
2418:     ZEND_ARG_OBJ_INFO(0, iterator, RecursiveIterator, 0)
2419: ZEND_END_ARG_INFO();
2420:
2421: static const zend_function_entry spl_funcs_RecursiveParentIterator[] = {
2422:     SPL_ME(RecursiveParentIterator, __construct, arginfo_parent_it__construct, ZEND_ACC_PUBLIC)
2423:     SPL_ME(RecursiveParentIterator, accept, RecursiveFilterIterator, hasChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2424:     PHP_FE_END
2425: };
2426:
2427:
2428: static const zend_function_entry spl_funcs_ParentIterator[] = {
2429:     SPL_ME(ParentIterator, __construct, arginfo_parent_it__construct, ZEND_ACC_PUBLIC)
2430:     SPL_ME(ParentIterator, accept, RecursiveFilterIterator, hasChildren, arginfo_recursive_it_void, ZEND_ACC_PUBLIC)
2431:     PHP_FE_END
2432: };
2433:
2434:
2435:
2436:
2437:
2438:
2439:
2440:
2441:
2442:
2443:
2444:
2445:
2446:
2447:
2448:
2449:
2450:
2451:
2452:
2453:
2454:
2455:
2456:
2457:
2458:
2459:
2460:
2461:
2462:
2463:
2464:
2465:
2466:
2467:
2468:
2469:
2470:
2471:
2472:
2473:
2474:
2475:
2476:
2477:
2478:
2479:
2480:
2481:
2482:
2483:
2484:
2485:
2486:
2487:
2488:
2489:
2490:
2491:
2492:
2493:
2494:
2495:
2496:
2497:
2498:
2499:
2500:
2501:
2502:
2503:
2504:
2505:
2506:
2507:
2508:
2509:
2510:
2511:
2512:
2513:
2514:
2515:
2516:
2517:
2518:
2519:
2520:
2521:
2522:
2523:
2524:
2525:
2526:
2527:
2528:
2529:
2530:
2531:
2532:
2533:
2534:
2535:
2536:
2537:
2538:
2539:
2540:
2541:
2542:
2543:
2544:
2545:
2546:
2547:
2548:
2549:
2550:
2551:
2552:
2553:
2554:
2555:
2556:
2557:
2558:
2559:
2560:
2561:
2562:
2563:
2564:
2565:
2566:
2567:
2568:
2569:
2570:
2571:
2572:
2573:
2574:
2575:
2576:
2577:
2578:
2579:
2580:
2581:
2582:
2583:
2584:
2585:
2586:
2587:
2588:
2589:
2590:
2591:
2592:
2593:
2594:
2595:
2596:
2597:
2598:
2599:
2600:
2601:
2602:
2603:
2604:
2605:
2606:
2607:
2608:
2609:
2610:
2611:
2612:
2613:
2614:
2615:
2616:
2617:
2618:
2619:
2620:
2621:
2622:
2623:
2624:
2625:
2626:
2627:
2628:
2629:
2630:
2631:
2632:
2633:
2634:
2635:
2636:
2637:
2638:
2639:
2640:
2641:
2642:
2643:
2644:
2645:
2646:
2647:
2648:
2649:
2650:
2651:
2652:
2653:
2654:
2655:
2656:
2657:
2658:
2659:
2660:
2661:
2662:
2663:
2664:
2665:
2666:
2667:
2668:
2669:
2670:
2671:
2672:
2673:
2674:
2675:
2676:
2677:
2678:
2679:
2680:
2681:
2682:
2683:
2684:
2685:
2686:
2687:
2688:
2689:
2690:
2691:
2692:
2693:
2694:
2695:
2696:
2697:
2698:
2699:
2700:
2701:
2702:
2703:
2704:
2705:
2706:
2707:
2708:
2709:
2710:
2711:
2712:
2713:
2714:
2715:
2716:
2717:
2718:
2719:
2720:
2721:
2722:
2723:
2724:
2725:
2726:
2727:
2728:
2729:
2730:
2731:
2732:
2733:
2734:
2735:
2736:
2737:
2738:
2739:
2740:
2741:
2742:
2743:
2744:
2745:
2746:
2747:
2748:
2749:
2750:
2751:
2752:
2753:
2754:
2755:
2756:
2757:
2758:
2759:
2760:
2761:
2762:
2763:
2764:
2765:
2766:
2767:
2768:
2769:
2770:
2771:
2772:
2773:
2774:
2775:
2776:
2777:
2778:
2779:
2780:
2781:
2782:
2783:
2784:
2785:
2786:
2787:
2788:
2789:
2790:
2791:
2792:
2793:
2794:
2795:
2796:
2797:
2798:
2799:
2800:
2801:
2802:
2803:
2804:
2805:
2806:
2807:
2808:
2809:
2810:
2811:
2812:
2813:
2814:
2815:
2816:
2817:
2818:
2819:
2820:
2821:
2822:
2823:
2824:
2825:
2826:
2827:
2828:
2829:
2830:
2831:
2832:
2833:
2834:
2835:
2836:
2837:
2838:
2839:
2840:
2841:
2842:
2843:
2844:
2845:
2846:
2847:
2848:
2849:
2850:
2851:
2852:
2853:
2854:
2855:
2856:
2857:
2858:
2859:
2860:
2861:
2862:
2863:
2864:
2865:
2866:
2867:
2868:
2869:
2870:
2871:
2872:
2873:
2874:
2875:
2876:
2877:
2878:
2879:
2880:
2881:
2882:
2883:
2884:
2885:
2886:
2887:
2888:
2889:
2890:
2891:
2892:
2893:
2894:
2895:
2896:
2897:
2898:
2899:
2900:
2901:
2902:
2903:
2904:
2905:
2906:
2907:
2908:
2909:
2910:
2911:
2912:
2913:
2914:
2915:
2916:
2917:
2918:
2919:
2920:
2921:
2922:
2923:
2924:
2925:
2926:
2927:
2928:
2929:
2930:
2931:
2932:
2933:
2934:
2935:
2936:
2937:
2938:
2939:
2940:
2941:
2942:
2943:
2944:
2945:
2946:
2947:
2948:
2949:
2950:
2951:
2952:
2953:
2954:
2955:
2956:
2957:
2958:
2959:
2960:
2961:
2962:
2963:
2964:
2965:
2966:
2967:
2968:
2969:
2970:
2971:
2972:
2973:
2974:
2975:
2976:
2977:
2978:
2979:
2980:
2981:
2982:
2983:
2984:
2985:
2986:
2987:
2988:
2989:
2990:
2991:
2992:
2993:
2994:
2995:
2996:
2997:
2998:
2999:
3000:
3001:
3002:
3003:
3004:
3005:
3006:
3007:
3008:
3009:
3010:
3011:
3012:
3013:
3014:
3015:
3016:
3017:
3018:
3019:
3020:
3021:
3022:
3023:
3024:
3025:
3026:
3027:
3028:
3029:
3030:
3031:
3032:
3033:
3034:
3035:
3036:
3037:
3038:
3039:
3040:
3041:
3042:
3043:
3044:
3045:
3046:
3047:
3048:
3049:
3050:
3051:
3052:
3053:
3054:
3055:
3056:
3057:
3058:
3059:
3060:
3061:
3062:
3063:
3064:
3065:
3066:
3067:
3068:
3069:
3070:
3071:
3072:
3073:
3074:
3075:
3076:
3077:
3078:
3079:
3080:
3081:
3082:
3083:
3084:
3085:
3086:
3087:
3088:
3089:
3090:
3091:
3092:
3093:
3094:
3095:
3096:
3097:
3098:
3099:
3100:
3101:
3102:
3103:
3104:
3105:
3106:
3107:
3108:
3109:
3110:
3111:
3112:
3113:
3114:
3115:
3116:
3117:
3118:
3119:
3120:
3121:
3122:
3123:
3124:
3125:
3126:
3127:
3128:
3129:
3130:
3131:
3132:
3133:
3134:
3135:
3136:
3137:
3138:
3139:
3140:
3141:
3142:
3143:
3144:
3145:
3146:
3147:
3148:
3149:
3150:
3151:
3152:
3153:
3154:
3155:
3156:
3157:
3158:
3159:
3160:
3161:
3162:
3163:
3164:
3165:
3166:
3167:
3168:
3169:
3170:
3171:
3172:
3173:
3174:
3175:
3176:
3177:
3178:
3179:
3180:
3181:
3182:
3183:
3184:
3185:
3186:
3187:
3188:
3189:
3190:
3191:
3192:
3193:
3194:
3195:
3196:
3197:
3198:
3199:
3200:
3201:
3202:
3203:
3204:
3205:
3206:
3207:
3208:
3209:
3210:
3211:
3212:
3213:
3214:
3215:
3216:
3217:
3218:
3219:
3220:
3221:
3222:
3223:
3224:
3225:
3226:
3227:
3228:
3229:
3230:
3231:
3232:
3233:
3234:
3235:
3236:
3237:
3238:
3239:
3240:
3241:
3242:
3243:
3244:
3245:
3246:
3247:
3248:
3249:
3250:
3251:
3252:
3253:
3254:
3255:
3256:
3257:
3258:
3259:
3260:
3261:
3262:
3263:
3264:
3265:
3266:
3267:
3268:
3269:
3270:
3271:
3272:
3273:
3274:
3275:
3276:
3277:
3278:
3279:
3280:
3281:
3282:
3283:
3284:
3285:
3286:
3287:
3288:
3289:
3290:
3291:
3292:
3293:
3294:
3295:
3296:
3297:
3298:
3299:
3300:
3301:
3302:
3303:
3304:
3305:
3306:
3307:
3308:
3309:
3310:
3311:
3312:
3313:
3314:
3315:
3316:
3317:
3318:
3319:
3320:
3321:
3322:
3323:
3324:
3325:
3326:
3327:
3328:
3329:
3330:
3331:
3332:
3333:
3334:
3335:
3336:
3337:
3338:
3339:
3340:
3341:
3342:
3343:
3344:
3345:
3346:
3347:
3348:
3349:
3350:
3351:
3352:
3353:
3354:
3355:
3356:
3357:
3358:
3359:
3360:
3361:
3362:
3363:
3364:
3365:
3366:
3367:
3368:
3369:
3370:
3371:
3372:
3373:
3374:
3375:
3376:
3377:
3378:
3379:
3380:
3381:
3382:
3383:
3384:
3385:
3386:
3387:
3388:
3389:
3390:
3391:
3392:
3393:
3394:
3395:
3396:
3397:
3398:
3399:
3400:
3401:
3402:
3403:
3404:
3405:
3406:
3407:
3408:
3409:
3410:
3411:
3412:
3413:
3414:
3415:
3416:
3417:
3418:
3419:
3420:
3421:
3422:
3423:
3424:
3425:
3426:
3427:
3428:
3429:
3430:
3431:
3432:
3433:
3434:
3435:
3436:
3437:
3438:
3439:
3440:
3441:
3442:
3443:
3444:
3445:
3446:
3447:
3448:
3449:
3450:
3451:
3452:
3453:
3454:
3455:
3456:
3457:
3458:
3459:
3460:
3461:
3462:
3463:
3464:
3465:
3466:
3467:
3468:
3469:
3470:
3471:
3472:
3473:
3474:
3475:
3476:
3477:
3478:
3479:
3480:
3481:
3482:
3483:
3484:
3485:
3486:
3487:
3488:
3489:
3490:
3491:
3492:
3493:
3494:
3495:
3496:
3497:
3498:
3499:
3500:
3501:
3502:
3503:
3504:
3505:
3506:
3507:
3508:
3509:
3510:
3511:
3512:
3513:
3514:
3515:
3516:
3517:
3518:
3519:
3520:
3521:
3522:
3523:
3524:
3525:
3526:
3527:
3528:
3529:
3530:
3531:
3532:
3533:
3534:
3535:
3536:
3537:
3538:
3539:
3540:
3541:
3542:
3543:
3544:
3545:
3546:
3547:
3548:
3549:
3550:
3551:
3552:
3553:
3554:
3555:
3556:
3557:
3558:
3559:
3560:
3561:
3562:
3563:
3564:
3565:
3566:
3567:
3568:
3569:
3570:
3571:
3572:
3573:
3574:
3575:
3576:
3577:
3578:
3579:
3580:
3581:
3582:
3583:
3584:
3585:
3586:
3587:
3588:
3589:
3590:
3591:
3592:
3593:
3594:
3595:
3596:
3597:
3598:
3599:
3600:
3601:
3602:
3603:
3604:
3605:
3606:
3607:
3608:
3609:
3610:
3611:
3612:
3613:
3614:
3615:
3616:
3617:
3618:
3619:
3620:
3621:
3622:
3623:
3624:
3625:
3626:
3627:
3628:
3629:
3630:
3631:
3632:
3633:
3634:
3635:
3636:
3637:
3638:
3639:
3640:
3641:
3642:
3643:
3644:
3645:
3646:
3647:
3648:
3649:
3650:
3651:
3652:
3653:
3654:
3655:
3656:
3657:
3658:
3659:
3660:
3661:
3662:
3663:
3664:
3665:
3666:
3667:
3668:
3669:
3670:
3671:
3672:
3673:
3674:
3675:
3676:
3677:
3678:
3679:
3680:
3681:
3682:
3683:
3684:
3685:
3686:
3687:
3688:
3689:
3690:
3691:
3692:
3693:
3694:
3695:
3696:
3697:
3698:
3699:
3700:
3701:
3702:
3703:
3704:
3705:
3706:
3707:
3708:
3709:
3710:
3711:
3712:
3713:
3714:
3715:
3716:
3717:
3718:
3719:
3720:
3721:
3722:
3723:
3724:
3725:
3726:
3727:
3728:
3729:
3730:
3731:
3732:
3733:
3734:
3735:
3736:
3737:
3738:
3739:
3740:
3741:
3742:
3743:
3744:
3745:
3746:
3747:
3748:
3749:
3750:
3751:
3752:
3753:
3754:
3755:
3756:
3757:
3758:
3759:
3760:
3761:
3762:
3763:
3764:
3765:
3766:
3767:
3768:
3769:
3770:
3771:
3772:
3773:
3774:
3775:
3776:
3777:
3778:
3779:
3780:
3781:
3782:
3783:
3784:
3785:
3786:
3787:
3788:
3789:
3790:
3791:
3792:
3793:
3794:
3795:
3796:
3797:
3798:
3799:
3800:
3801:
3802:
3803:
3804:
3805:
3806:
3807:
3808:
3809:
3810:
3811:
3812:
3813:
3814:
3815:
3816:
3817:
3818:
3819:
3820:
3821:
3822:
3823:
3824:
3825:
3826:
3827:
3828:
3829:
3830:
3831:
3832:
3833:
3834:
3835:
3836:
3837:
3838:
3839:
3840:
3841:
3842:
3843:
3844:
3845:
3846:
3847:
3848:
3849:
3850:
3851:
3852:
3853:
3854:
3855:
3856:
3857:
3858:
3859:
3860:
3861:
3862:
3863:
3864:
3865:
3866:
3867:
3868:
3869:
3870:
3871:
3872:
3873:
3874:
3875:
3876:
3877:
3878:
3879:
3880:
3881:
3882:
3883:
3884:
3885:
3886:
3887:
3888:
3889:
3890:
3891:
3892:
3893:
3894:
3895:
3896:
3897:
3898:
3899:
3900:
3901:
3902:
3903:
3904:
3905:
3906:
3907:
3908:
3909:
3910:
3911:
3912:
3913:
3914:
3915:
3916:
3917:
3918:
3919:
3920:
3921:
3922:
3923:
3924:
3925:
3926:
3927:
3928:
3929:
3930:
3931:
3932:
3933:
3934:
3935:
3936:
3937:
3938:
3939:
3940:
3941:
3942:
3943:
3944:
3945:
3946:
3947:
3948:
3949:
3950:
3951:
3952:
3953:
3954:
3955:
3956:
3957:
3958:
3959:
3960:
3961:
3962:
3963:
3964:
3965:
3966:
3967:
3968:
3969:
3970:
3971:
3972:
3973:
3974:
3975:
3976:
3977:
3978:
3979:
3980:
3981:
3982:
3983:
3984:
3985:
3986:
3987:
3988:
3989:
3990:
3991:
3992:
3993:
3994:
3995:
3996:
3997:
3998:
3999:
4000:

```

```
getThis()->name));
```



```

2989: // Number of cached elements */
2990: SPL_METHOD(CachingIterator, count)
2991: {
2992:     spl_dual_it_obj_t *intern;
2993:     if (zend_parse_parameters_none() == FAILURE) {
2994:         return;
2995:     }
2996:
2997:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
2998:
2999:     if (! (intern->u.caching.flags & CIT_FULL_CACHE)) {
3000:         zend_throw_exception(spl_ce_BadMethodCallException, 0, "Is does not use a full cache (see CachingIterator::__construct)", ZSTR_VAL(Z_STRCP_P(getThis())->name));
3001:         return;
3002:     }
3003:
3004:     RETURN_LONG(zend_hash_num_elements(Z_ARRVAL(intern->u.caching.cache)));
3005: }
3006: /* }}} */
3007:
3008: ZEND_BEGIN_ARG_INFO_EX(arginfo_caching_it___construct, 0, 0, 1)
3009:     ZEND_ARG_CBL_INFO(0, iterator, iterator, 0)
3010:     ZEND_ARG_INFO(0, flags)
3011: ZEND_END_ARG_INFO();
3012:
3013: ZEND_BEGIN_ARG_INFO(arginfo_caching_it_setFlags, 0)
3014:     ZEND_ARG_INFO(0, flags)
3015: ZEND_END_ARG_INFO();
3016:
3017: ZEND_BEGIN_ARG_INFO(arginfo_caching_it_offsetGet, 0)
3018:     ZEND_ARG_INFO(0, index)
3019: ZEND_END_ARG_INFO();
3020:
3021: ZEND_BEGIN_ARG_INFO(arginfo_caching_it_offsetSet, 0)
3022:     ZEND_ARG_INFO(0, index)
3023:     ZEND_ARG_INFO(0, newval)
3024: ZEND_END_ARG_INFO();
3025:
3026: static const zend_function_entry spl_func_CachingIterator[] = {
3027:     SPL_ME(CachingIterator, __construct,      arginfo_caching_it___construct, ZEND_ACC_PUBLIC)
3028:     SPL_ME(CachingIterator, rewind,          arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3029:     SPL_ME(CachingIterator, valid,           arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3030:     SPL_ME(dual_it, key,                    arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3031:     SPL_ME(dual_it, current,               arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3032:     SPL_ME(CachingIterator, next,           arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3033:     SPL_ME(CachingIterator, hasNext,       arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3034:     SPL_ME(CachingIterator, __toString,     arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3035:     SPL_ME(dual_it, getInnerIterator,       arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3036:     SPL_ME(CachingIterator, getFlags,       arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3037:     SPL_ME(CachingIterator, setFlags,       arginfo_caching_it_setFlags,  ZEND_ACC_PUBLIC)
3038:     SPL_ME(CachingIterator, offsetGet,      arginfo_caching_it_offsetGet, ZEND_ACC_PUBLIC)
3039:     SPL_ME(CachingIterator, offsetSet,      arginfo_caching_it_offsetSet, ZEND_ACC_PUBLIC)
3040:     SPL_ME(CachingIterator, offsetUnset,    arginfo_caching_it_offsetGet, ZEND_ACC_PUBLIC)
3041:     SPL_ME(CachingIterator, offsetExists,   arginfo_caching_it_offsetGet, ZEND_ACC_PUBLIC)
3042:     SPL_ME(CachingIterator, getCache,       arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3043:     SPL_ME(CachingIterator, count,          arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3044:     PHP_FE_END
3045: };
3046:
3047: /* {{{ proto void RecursiveCachingIterator::__construct(RecursiveIterator $it, $flags = CIT_CALL_TOSTRING)
3048:    Create an iterator from a RecursiveIterator */
3049: SPL_METHOD(RecursiveCachingIterator, __construct)
3050: {
3051:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_RecursiveCachingIterator, spl_ce_RecursiveIterator, DIT_RecursiveCachingIterator);
3052: } /* }}} */
3053:
3054: /* {{{ proto bool RecursiveCachingIterator::hasChildren()
3055:    Check whether the current element of the inner iterator has children */
3056: SPL_METHOD(RecursiveCachingIterator, hasChildren)
3057: {
3058:     spl_dual_it_obj_t *intern;
3059:
3060:     if (zend_parse_parameters_none() == FAILURE) {
3061:         return;
3062:     }
3063:
3064:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3065:
3066:     RETURN_BOOL(! (TYPE(intern->u.caching.children) != IS_UNDEF));
3067: } /* }}} */
3068:
3069: /* {{{ proto RecursiveCachingIterator RecursiveCachingIterator::getChildren()
3070:    Return the inner iterator's children as a RecursiveCachingIterator */
3071: SPL_METHOD(RecursiveCachingIterator, getChildren)
3072: {
3073:     spl_dual_it_obj_t *intern;
3074:
3075:     if (zend_parse_parameters_none() == FAILURE) {
3076:         return;
3077:     }
3078:
3079:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3080:
3081:     if (Z_TYPE(intern->u.caching.children) != IS_UNDEF) {
3082:         zval *value = &intern->u.caching.children;
3083:
3084:         ZVAL_DEREF(value);
3085:         ZVAL_COPY(&return_value, value);
3086:     } else {
3087:         RETURN_NULL();
3088:     }
3089: } /* }}} */
3090:
3091: ZEND_BEGIN_ARG_INFO_EX(arginfo_caching_rec_it___construct, 0, ZEND_RETURN_VALUE, 1)
3092:     ZEND_ARG_CBL_INFO(0, iterator, iterator, 0)
3093:     ZEND_ARG_INFO(0, flags)
3094: ZEND_END_ARG_INFO();
3095:
3096: static const zend_function_entry spl_func_RecursiveCachingIterator[] = {
3097:     SPL_ME(RecursiveCachingIterator, __construct,      arginfo_caching_rec_it___construct, ZEND_ACC_PUBLIC)
3098:     SPL_ME(RecursiveCachingIterator, hasChildren,     arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3099:     SPL_ME(RecursiveCachingIterator, getChildren,     arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3100:     PHP_FE_END
3101: };
3102:
3103: /* {{{ proto void IteratorIterator::__construct(Traversable $it)
3104:    Create an iterator from anything that is traversable */
3105: SPL_METHOD(IteratorIterator, __construct)
3106: {
3107:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_IteratorIterator, zend_ce_traversable, DIT_IteratorIterator);
3108: } /* }}} */
3109:
3110: ZEND_BEGIN_ARG_INFO(arginfo_iterator_it___construct, 0)
3111:     ZEND_ARG_CBL_INFO(0, iterator, traversable, 0)
3112: ZEND_END_ARG_INFO();
3113:
3114: static const zend_function_entry spl_func_IteratorIterator[] = {
3115:     SPL_ME(IteratorIterator, __construct,      arginfo_iterator_it___construct, ZEND_ACC_PUBLIC)
3116:     SPL_ME(dual_it, rewind,                   arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3117:     SPL_ME(dual_it, valid,                   arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3118:     SPL_ME(dual_it, key,                    arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3119:     SPL_ME(dual_it, current,               arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3120:     SPL_ME(dual_it, next,                  arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3121:     SPL_ME(dual_it, getInnerIterator,       arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3122:     PHP_FE_END
3123: };
3124:
3125: /* {{{ proto void NoRewindIterator::__construct(Iterator $it)
3126:    Create an iterator from another iterator */
3127: SPL_METHOD(NoRewindIterator, __construct)
3128: {
3129:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_NoRewindIterator, zend_ce_iterator, DIT_NoRewindIterator);
3130: } /* }}} */
3131:
3132: /* {{{ proto void NoRewindIterator::rewind()
3133:    Prevent a call to inner iterators rewind() */
3134: SPL_METHOD(NoRewindIterator, rewind)
3135: {
3136:     if (zend_parse_parameters_none() == FAILURE) {
3137:         return;
3138:     }
3139:
3140:     /* nothing to do */
3141: } /* }}} */
3142:
3143: /* {{{ proto bool NoRewindIterator::valid()
3144:    Return inner iterator's valid() */
3145: SPL_METHOD(NoRewindIterator, valid)
3146: {
3147:     spl_dual_it_obj_t *intern;
3148:
3149:     if (zend_parse_parameters_none() == FAILURE) {
3150:         return;
3151:     }
3152:
3153:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3154:
3155:     if (intern->inner.iterator->funcs->get_current_key(intern->inner.iterator, return_value);
3156:         ) else {
3157:             RETURN_NULL();
3158:         }
3159: } /* }}} */
3160:
3161: /* {{{ proto mixed NoRewindIterator::key()
3162:    Return inner iterators key() */
3163: SPL_METHOD(NoRewindIterator, key)
3164: {
3165:     spl_dual_it_obj_t *intern;
3166:
3167:     if (zend_parse_parameters_none() == FAILURE) {
3168:         return;
3169:     }
3170:
3171:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3172:
3173:     if (intern->inner.iterator->funcs->get_current_key(intern->inner.iterator, return_value);
3174:         ) else {
3175:             RETURN_NULL();
3176:         }
3177: } /* }}} */
3178:
3179: /* {{{ proto mixed NoRewindIterator::next()
3180:    Return inner iterators next() */
3181: SPL_METHOD(NoRewindIterator, next)
3182: {
3183:     spl_dual_it_obj_t *intern;
3184:
3185:     if (zend_parse_parameters_none() == FAILURE) {
3186:         return;
3187:     }
3188:
3189:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3190:
3191:     if (intern->inner.iterator->funcs->move_forward(intern->inner.iterator);
3192:         ) else {
3193:             RETURN_NULL();
3194:         }
3195: } /* }}} */
3196:
3197: static const zend_function_entry spl_func_NoRewindIterator[] = {
3198:     SPL_ME(NoRewindIterator, __construct,      arginfo_no_rewind_it___construct, ZEND_ACC_PUBLIC)
3199:     SPL_ME(dual_it, rewind,                   arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3200:     SPL_ME(dual_it, valid,                   arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3201:     SPL_ME(dual_it, key,                    arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3202:     SPL_ME(dual_it, current,               arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3203:     SPL_ME(dual_it, next,                  arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3204:     SPL_ME(dual_it, getInnerIterator,       arginfo_recursive_it_void,    ZEND_ACC_PUBLIC)
3205:     PHP_FE_END
3206: };
3207:
3208: /* {{{ proto void InfiniteIterator::__construct(Iterator $it)
3209:    Create an iterator from another iterator */
3210: SPL_METHOD(InfiniteIterator, __construct)
3211: {
3212:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_InfiniteIterator, zend_ce_iterator, DIT_InfiniteIterator);
3213: } /* }}} */
3214:
3215: /* {{{ proto void InfiniteIterator::next()
3216:    Prevent a call to inner iterators rewind() (internally the current data will be fetched if valid) */
3217: SPL_METHOD(InfiniteIterator, next)
3218: {
3219:     spl_dual_it_obj_t *intern;
3220:
3221:     if (zend_parse_parameters_none() == FAILURE) {
3222:         return;
3223:     }
3224:
3225:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3226:
3227:     if (zend_parse_parameters_none() == FAILURE) {
3228:         return;
3229:     }
3230:
3231:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3232:
3233:     if (zend_parse_parameters_none() == FAILURE) {
3234:         return;
3235:     }
3236:
3237:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3238:
3239:     if (zend_parse_parameters_none() == FAILURE) {
3240:         return;
3241:     }
3242:
3243:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3244:
3245:     if (zend_parse_parameters_none() == FAILURE) {
3246:         return;
3247:     }
3248:
3249:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3250:
3251:     if (zend_parse_parameters_none() == FAILURE) {
3252:         return;
3253:     }
3254:
3255:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3256:
3257:     if (zend_parse_parameters_none() == FAILURE) {
3258:         return;
3259:     }
3260:
3261:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3262:
3263:     if (zend_parse_parameters_none() == FAILURE) {
3264:         return;
3265:     }
3266:
3267:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3268:
3269:     if (zend_parse_parameters_none() == FAILURE) {
3270:         return;
3271:     }
3272:
3273:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3274:
3275:     if (zend_parse_parameters_none() == FAILURE) {
3276:         return;
3277:     }
3278:
3279:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3280:
3281:     if (zend_parse_parameters_none() == FAILURE) {
3282:         return;
3283:     }
3284:
3285:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3286:
3287:     if (zend_parse_parameters_none() == FAILURE) {
3288:         return;
3289:     }
3290:
3291:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3292:
3293:     if (zend_parse_parameters_none() == FAILURE) {
3294:         return;
3295:     }
3296:
3297:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3298:
3299:     if (zend_parse_parameters_none() == FAILURE) {
3300:         return;
3301:     }
3302:
3303:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3304:
3305:     if (zend_parse_parameters_none() == FAILURE) {
3306:         return;
3307:     }
3308:
3309:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3310:
3311:     if (zend_parse_parameters_none() == FAILURE) {
3312:         return;
3313:     }
3314:
3315:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3316:
3317:     if (zend_parse_parameters_none() == FAILURE) {
3318:         return;
3319:     }
3320:
3321:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3322:
3323:     if (zend_parse_parameters_none() == FAILURE) {
3324:         return;
3325:     }
3326:
3327:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3328:
3329:     if (zend_parse_parameters_none() == FAILURE) {
3330:         return;
3331:     }
3332:
3333:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3334:
3335:     if (zend_parse_parameters_none() == FAILURE) {
3336:         return;
3337:     }
3338:
3339:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3340:
3341:     if (zend_parse_parameters_none() == FAILURE) {
3342:         return;
3343:     }
3344:
3345:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3346:
3347:     if (zend_parse_parameters_none() == FAILURE) {
3348:         return;
3349:     }
3350:
3351:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3352:
3353:     if (zend_parse_parameters_none() == FAILURE) {
3354:         return;
3355:     }
3356:
3357:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3358:
3359:     if (zend_parse_parameters_none() == FAILURE) {
3360:         return;
3361:     }
3362:
3363:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3364:
3365:     if (zend_parse_parameters_none() == FAILURE) {
3366:         return;
3367:     }
3368:
3369:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3370:
3371:     if (zend_parse_parameters_none() == FAILURE) {
3372:         return;
3373:     }
3374:
3375:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3376:
3377:     if (zend_parse_parameters_none() == FAILURE) {
3378:         return;
3379:     }
3380:
3381:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3382:
3383:     if (zend_parse_parameters_none() == FAILURE) {
3384:         return;
3385:     }
3386:
3387:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3388:
3389:     if (zend_parse_parameters_none() == FAILURE) {
3390:         return;
3391:     }
3392:
3393:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3394:
3395:     if (zend_parse_parameters_none() == FAILURE) {
3396:         return;
3397:     }
3398:
3399:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3400:
3401:     if (zend_parse_parameters_none() == FAILURE) {
3402:         return;
3403:     }
3404:
3405:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3406:
3407:     if (zend_parse_parameters_none() == FAILURE) {
3408:         return;
3409:     }
3410:
3411:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3412:
3413:     if (zend_parse_parameters_none() == FAILURE) {
3414:         return;
3415:     }
3416:
3417:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3418:
3419:     if (zend_parse_parameters_none() == FAILURE) {
3420:         return;
3421:     }
3422:
3423:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3424:
3425:     if (zend_parse_parameters_none() == FAILURE) {
3426:         return;
3427:     }
3428:
3429:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3430:
3431:     if (zend_parse_parameters_none() == FAILURE) {
3432:         return;
3433:     }
3434:
3435:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3436:
3437:     if (zend_parse_parameters_none() == FAILURE) {
3438:         return;
3439:     }
3440:
3441:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3442:
3443:     if (zend_parse_parameters_none() == FAILURE) {
3444:         return;
3445:     }
3446:
3447:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3448:
3449:     if (zend_parse_parameters_none() == FAILURE) {
3450:         return;
3451:     }
3452:
3453:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3454:
3455:     if (zend_parse_parameters_none() == FAILURE) {
3456:         return;
3457:     }
3458:
3459:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3460:
3461:     if (zend_parse_parameters_none() == FAILURE) {
3462:         return;
3463:     }
3464:
3465:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3466:
3467:     if (zend_parse_parameters_none() == FAILURE) {
3468:         return;
3469:     }
3470:
3471:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3472:
3473:     if (zend_parse_parameters_none() == FAILURE) {
3474:         return;
3475:     }
3476:
3477:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3478:
3479:     if (zend_parse_parameters_none() == FAILURE) {
3480:         return;
3481:     }
3482:
3483:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3484:
3485:     if (zend_parse_parameters_none() == FAILURE) {
3486:         return;
3487:     }
3488:
3489:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3490:
3491:     if (zend_parse_parameters_none() == FAILURE) {
3492:         return;
3493:     }
3494:
3495:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3496:
3497:     if (zend_parse_parameters_none() == FAILURE) {
3498:         return;
3499:     }
3500:
3501:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3502:
3503:     if (zend_parse_parameters_none() == FAILURE) {
3504:         return;
3505:     }
3506:
3507:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3508:
3509:     if (zend_parse_parameters_none() == FAILURE) {
3510:         return;
3511:     }
3512:
3513:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3514:
3515:     if (zend_parse_parameters_none() == FAILURE) {
3516:         return;
3517:     }
3518:
3519:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3520:
3521:     if (zend_parse_parameters_none() == FAILURE) {
3522:         return;
3523:     }
3524:
3525:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3526:
3527:     if (zend_parse_parameters_none() == FAILURE) {
3528:         return;
3529:     }
3530:
3531:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3532:
3533:     if (zend_parse_parameters_none() == FAILURE) {
3534:         return;
3535:     }
3536:
3537:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3538:
3539:     if (zend_parse_parameters_none() == FAILURE) {
3540:         return;
3541:     }
3542:
3543:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3544:
3545:     if (zend_parse_parameters_none() == FAILURE) {
3546:         return;
3547:     }
3548:
3549:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3550:
3551:     if (zend_parse_parameters_none() == FAILURE) {
3552:         return;
3553:     }
3554:
3555:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3556:
3557:     if (zend_parse_parameters_none() == FAILURE) {
3558:         return;
3559:     }
3560:
3561:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3562:
3563:     if (zend_parse_parameters_none() == FAILURE) {
3564:         return;
3565:     }
3566:
3567:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3568:
3569:     if (zend_parse_parameters_none() == FAILURE) {
3570:         return;
3571:     }
3572:
3573:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3574:
3575:     if (zend_parse_parameters_none() == FAILURE) {
3576:         return;
3577:     }
3578:
3579:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3580:
3581:     if (zend_parse_parameters_none() == FAILURE) {
3582:         return;
3583:     }
3584:
3585:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3586:
3587:     if (zend_parse_parameters_none() == FAILURE) {
3588:         return;
3589:     }
3590:
3591:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3592:
3593:     if (zend_parse_parameters_none() == FAILURE) {
3594:         return;
3595:     }
3596:
3597:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3598:
3599:     if (zend_parse_parameters_none() == FAILURE) {
3600:         return;
3601:     }
3602:
3603:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3604:
3605:     if (zend_parse_parameters_none() == FAILURE) {
3606:         return;
3607:     }
3608:
3609:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3610:
3611:     if (zend_parse_parameters_none() == FAILURE) {
3612:         return;
3613:     }
3614:
3615:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3616:
3617:     if (zend_parse_parameters_none() == FAILURE) {
3618:         return;
3619:     }
3620:
3621:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3622:
3623:     if (zend_parse_parameters_none() == FAILURE) {
3624:         return;
3625:     }
3626:
3627:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3628:
3629:     if (zend_parse_parameters_none() == FAILURE) {
3630:         return;
3631:     }
3632:
3633:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3634:
3635:     if (zend_parse_parameters_none() == FAILURE) {
3636:         return;
3637:     }
3638:
3639:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3640:
3641:     if (zend_parse_parameters_none() == FAILURE) {
3642:         return;
3643:     }
3644:
3645:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3646:
3647:     if (zend_parse_parameters_none() == FAILURE) {
3648:         return;
3649:     }
3650:
3651:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3652:
3653:     if (zend_parse_parameters_none() == FAILURE) {
3654:         return;
3655:     }
3656:
3657:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3658:
3659:     if (zend_parse_parameters_none() == FAILURE) {
3660:         return;
3661:     }
3662:
3663:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3664:
3665:     if (zend_parse_parameters_none() == FAILURE) {
3666:         return;
3667:     }
3668:
3669:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3670:
3671:     if (zend_parse_parameters_none() == FAILURE) {
3672:         return;
3673:     }
3674:
3675:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3676:
3677:     if (zend_parse_parameters_none() == FAILURE) {
3678:         return;
3679:     }
3680:
3681:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3682:
3683:     if (zend_parse_parameters_none() == FAILURE) {
3684:         return;
3685:     }
3686:
3687:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3688:
3689:     if (zend_parse_parameters_none() == FAILURE) {
3690:         return;
3691:     }
3692:
3693:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3694:
3695:     if (zend_parse_parameters_none() == FAILURE) {
3696:         return;
3697:     }
3698:
3699:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3700:
3701:     if (zend_parse_parameters_none() == FAILURE) {
3702:         return;
3703:     }
3704:
3705:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3706:
3707:     if (zend_parse_parameters_none() == FAILURE) {
3708:         return;
3709:     }
3710:
3711:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3712:
3713:     if (zend_parse_parameters_none() == FAILURE) {
3714:         return;
3715:     }
3716:
3717:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3718:
3719:     if (zend_parse_parameters_none() == FAILURE) {
3720:         return;
3721:     }
3722:
3723:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3724:
3725:     if (zend_parse_parameters_none() == FAILURE) {
3726:         return;
3727:     }
3728:
3729:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3730:
3731:     if (zend_parse_parameters_none() == FAILURE) {
3732:         return;
3733:     }
3734:
3735:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3736:
3737:     if (zend_parse_parameters_none() == FAILURE) {
3738:         return;
3739:     }
3740:
3741:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3742:
3743:     if (zend_parse_parameters_none() == FAILURE) {
3744:         return;
3745:     }
3746:
3747:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3748:
3749:     if (zend_parse_parameters_none() == FAILURE) {
3750:         return;
3751:     }
3752:
3753:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3754:
3755:     if (zend_parse_parameters_none() == FAILURE) {
3756:         return;
3757:     }
3758:
3759:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3760:
3761:     if (zend_parse_parameters_none() == FAILURE) {
3762:         return;
3763:     }
3764:
3765:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3766:
3767:     if (zend_parse_parameters_none() == FAILURE) {
3768:         return;
3769:     }
3770:
3771:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3772:
3773:     if (zend_parse_parameters_none() == FAILURE) {
3774:         return;
3775:     }
3776:
3777:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3778:
3779:     if (zend_parse_parameters_none() == FAILURE) {
3780:         return;
3781:     }
3782:
3783:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3784:
3785:     if (zend_parse_parameters_none() == FAILURE) {
3786:         return;
3787:     }
3788:
3789:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3790:
3791:     if (zend_parse_parameters_none() == FAILURE) {
3792:         return;
3793:     }
3794:
3795:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3796:
3797:     if (zend_parse_parameters_none() == FAILURE) {
3798:         return;
3799:     }
3800:
3801:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3802:
3803:     if (zend_parse_parameters_none() == FAILURE) {
3804:         return;
3805:     }
3806:
3807:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3808:
3809:     if (zend_parse_parameters_none() == FAILURE) {
3810:         return;
3811:     }
3812:
3813:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3814:
3815:     if (zend_parse_parameters_none() == FAILURE) {
3816:         return;
3817:     }
3818:
3819:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3820:
3821:     if (zend_parse_parameters_none() == FAILURE) {
3822:         return;
3823:     }
3824:
3825:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3826:
3827:     if (zend_parse_parameters_none() == FAILURE) {
3828:         return;
3829:     }
3830:
3831:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3832:
3833:     if (zend_parse_parameters_none() == FAILURE) {
3834:         return;
3835:     }
3836:
3837:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3838:
3839:     if (zend_parse_parameters_none() == FAILURE) {
3840:         return;
3841:     }
3842:
3843:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3844:
3845:     if (zend_parse_parameters_none() == FAILURE) {
3846:         return;
3847:     }
3848:
3849:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3850:
3851:     if (zend_parse_parameters_none() == FAILURE) {
3852:         return;
3853:     }
3854:
3855:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3856:
3857:     if (zend_parse_parameters_none() == FAILURE) {
3858:         return;
3859:     }
3860:
3861:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3862:
3863:     if (zend_parse_parameters_none() == FAILURE) {
3864:         return;
3865:     }
3866:
3867:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3868:
3869:     if (zend_parse_parameters_none() == FAILURE) {
3870:         return;
3871:     }
3872:
3873:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3874:
3875:     if (zend_parse_parameters_none() == FAILURE) {
3876:         return;
3877:     }
3878:
3879:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3880:
3881:     if (zend_parse_parameters_none() == FAILURE) {
3882:         return;
3883:     }
3884:
3885:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3886:
3887:     if (zend_parse_parameters_none() == FAILURE) {
3888:         return;
3889:     }
3890:
3891:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3892:
3893:     if (zend_parse_parameters_none() == FAILURE) {
3894:         return;
3895:     }
3896:
3897:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3898:
3899:     if (zend_parse_parameters_none() == FAILURE) {
3900:         return;
3901:     }
3902:
3903:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3904:
3905:     if (zend_parse_parameters_none() == FAILURE) {
3906:         return;
3907:     }
3908:
3909:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3910:
3911:     if (zend_parse_parameters_none() == FAILURE) {
3912:         return;
3913:     }
3914:
3915:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3916:
3917:     if (zend_parse_parameters_none() == FAILURE) {
3918:         return;
3919:     }
3920:
3921:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3922:
3923:     if (zend_parse_parameters_none() == FAILURE) {
3924:         return;
3925:     }
3926:
3927:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3928:
3929:     if (zend_parse_parameters_none() == FAILURE) {
3930:         return;
3931:     }
3932:
3933:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3934:
3935:     if (zend_parse_parameters_none() == FAILURE) {
3936:         return;
3937:     }
3938:
3939:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3940:
3941:     if (zend_parse_parameters_none() == FAILURE) {
3942:         return;
3943:     }
3944:
3945:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3946:
3947:     if (zend_parse_parameters_none() == FAILURE) {
3948:         return;
3949:     }
3950:
3951:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3952:
3953:     if (zend_parse_parameters_none() == FAILURE) {
3954:         return;
3955:     }
3956:
3957:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3958:
3959:     if (zend_parse_parameters_none() == FAILURE) {
3960:         return;
3961:     }
3962:
3963:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3964:
3965:     if (zend_parse_parameters_none() == FAILURE) {
3966:         return;
3967:     }
3968:
3969:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3970:
3971:     if (zend_parse_parameters_none() == FAILURE) {
3972:         return;
3973:     }
3974:
3975:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3976:
3977:     if (zend_parse_parameters_none() == FAILURE) {
3978:         return;
3979:     }
3980:
3981:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3982:
3983:     if (zend_parse_parameters_none() == FAILURE) {
3984:         return;
3985:     }
3986:
3987:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3988:
3989:     if (zend_parse_parameters_none() == FAILURE) {
3990:         return;
3991:     }
3992:
3993:     SPL_FETCH_AND_CHECK_DUAL_IT(intern, getThis());
3994:
3995:     if (zend_parse_parameters_none() == FAILURE) {
3996:        
```

```

3364: SPL_METHOD (AppendIterator, __construct,
3365: {
3366:     spl_dual_it_construct (INTERNAL_FUNCTION_PARAM_PASSTHRU, spl_ce_AppendIterator, zend_ce_iterator, DIT_AppendIterator);
3367: } /* }}} */
3368:
3369: /* {{{ proto void AppendIterator::append(Iterator $t)
3370:  * Append an Iterator */
3371: SPL_METHOD (AppendIterator, append)
3372: {
3373:     spl_dual_it_object *intern;
3374:     zval *it;
3375:
3376:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3377:
3378:     if (zend_parse_parameters_ex (ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS(), "O", &it, zend_ce_iterator) == FAILURE) {
3379:         return;
3380:     }
3381:
3382:     if (intern->u.append_iterator->funcs->valid (intern->u.append_iterator) == SUCCESS && spl_dual_it_valid (intern) != SUCCESS) {
3383:         spl_array_iterator_append (intern->u.append_array, it);
3384:     } else {
3385:         spl_array_iterator_append (intern->u.append_array, it);
3386:     }
3387:
3388:     if (intern->inner_iterator != spl_dual_it_valid (intern)) != SUCCESS {
3389:         if (intern->u.append_iterator->funcs->valid (intern->u.append_iterator) != SUCCESS) {
3390:             intern->u.append_iterator->funcs->rewind (intern->u.append_iterator);
3391:         }
3392:     } do {
3393:         spl_append_it_next_iterator (intern);
3394:     } while (IS_TRUE (intern->inner_subj) != Z_OBJ_P (it));
3395:     spl_append_it_fetch (intern);
3396: }
3397: } /* }}} */
3398:
3399: /* {{{ proto mixed AppendIterator::current()
3400:  * Get the current element value */
3401: SPL_METHOD (AppendIterator, current)
3402: {
3403:     spl_dual_it_object *intern;
3404:
3405:     if (zend_parse_parameters_none() == FAILURE) {
3406:         return;
3407:     }
3408:
3409:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3410:
3411:     spl_dual_it_fetch (intern, 1);
3412:     if (Z_TYPE (intern->current_data) != IS_UNDEF) {
3413:         zval *value = &intern->current_data;
3414:
3415:         ZVAL_DEREF (value);
3416:         ZVAL_COPY (return_value, value);
3417:     } else {
3418:         RETURN_NULL();
3419:     }
3420: } /* }}} */
3421:
3422: /* {{{ proto void AppendIterator::rewind()
3423:  * Rewind to the first iterator and rewind the first iterator, too */
3424: SPL_METHOD (AppendIterator, rewind)
3425: {
3426:     spl_dual_it_object *intern;
3427:
3428:     if (zend_parse_parameters_none() == FAILURE) {
3429:         return;
3430:     }
3431:
3432:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3433:
3434:     intern->u.append_iterator->funcs->rewind (intern->u.append_iterator);
3435:     if (spl_append_it_next_iterator (intern) == SUCCESS) {
3436:         spl_append_it_fetch (intern);
3437:     }
3438: } /* }}} */
3439:
3440: /* {{{ proto bool AppendIterator::valid()
3441:  * Check if the current state is valid */
3442: SPL_METHOD (AppendIterator, valid)
3443: {
3444:     spl_dual_it_object *intern;
3445:
3446:     if (zend_parse_parameters_none() == FAILURE) {
3447:         return;
3448:     }
3449:
3450:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3451:
3452:     RETURN_BOOL (Z_TYPE (intern->current_data) != IS_UNDEF);
3453: } /* }}} */
3454:
3455: /* {{{ proto void AppendIterator::next()
3456:  * Forward to next element */
3457: SPL_METHOD (AppendIterator, next)
3458: {
3459:     spl_dual_it_object *intern;
3460:
3461:     if (zend_parse_parameters_none() == FAILURE) {
3462:         return;
3463:     }
3464:
3465:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3466:
3467:     spl_append_it_next (intern);
3468: } /* }}} */
3469:
3470: /* {{{ proto int AppendIterator::getIteratorIndex()
3471:  * Get index of iterator */
3472: SPL_METHOD (AppendIterator, getIteratorIndex)
3473: {
3474:     spl_dual_it_object *intern;
3475:
3476:     if (zend_parse_parameters_none() == FAILURE) {
3477:         return;
3478:     }
3479:
3480:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3481:
3482:     APPEND_IT_CHECK_CTOR (intern);
3483:     spl_array_iterator_key (&intern->u.append_array, &return_value);
3484: } /* }}} */
3485:
3486: /* {{{ proto ArrayIterator AppendIterator::getArrayIterator()
3487:  * Get access to inner ArrayIterator */
3488: SPL_METHOD (AppendIterator, getArrayIterator)
3489: {
3490:     spl_dual_it_object *intern;
3491:     zval *value;
3492:
3493:     if (zend_parse_parameters_none() == FAILURE) {
3494:         return;
3495:     }
3496:
3497:     SPL_FETCH_AND_CHECK_DUAL_IT (intern, getThis());
3498:
3499:     value = &intern->u.append_array;
3500:     ZVAL_DEREF (value);
3501:     ZVAL_COPY (return_value, value);
3502: } /* }}} */
3503:
3504: ZEND_BEGIN_ARG_INFO (arginfo_append_it_append, 0)
3505: ZEND_ARG_OBJ_INFO (0, iterator, Iterator, 0)
3506: ZEND_END_ARG_INFO ()
3507:
3508: static const zend_function_entry spl_funcs_AppendIterator[] = {
3509:     SPL_ME (AppendIterator, __construct, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3510:     SPL_ME (AppendIterator, append, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3511:     SPL_ME (AppendIterator, rewind, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3512:     SPL_ME (AppendIterator, valid, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3513:     SPL_ME (Dual_it, key, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3514:     SPL_ME (AppendIterator, current, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3515:     SPL_ME (AppendIterator, next, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3516:     SPL_ME (Dual_it, getInnerIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3517:     SPL_ME (AppendIterator, getIteratorIndex, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3518:     SPL_ME (AppendIterator, getArrayIterator, arginfo_recursive_it_void, ZEND_ACC_PUBLIC),
3519:     PHP_FE_END
3520: };
3521:
3522: PHP_API int spl_iterator_apply (zval *obj, spl_iterator_apply_func_t apply_func, void *puser)
3523: {
3524:     zend_object_iterator *iter;
3525:     zend_class_entry *ce = Z_OBJCE_P (obj);
3526:
3527:     iter = ce->get_iterator (ce, obj, 0);
3528:
3529:     if (EG (exception)) {
3530:         goto done;
3531:     }
3532:
3533:     iter->index = 0;
3534:     if (iter->funcs->rewind() {
3535:         iter->funcs->rewind (iter);
3536:     } if (EG (exception)) {
3537:         goto done;
3538:     }
3539:
3540:     while (iter->funcs->valid (iter) == SUCCESS) {
3541:         if (EG (exception)) {
3542:             goto done;
3543:         }
3544:
3545:         if (apply_func (iter, puser) == ZEND_HASH_APPLY_STOP || EG (exception)) {
3546:             goto done;
3547:         }
3548:         iter->index++;
3549:         iter->funcs->move_forward (iter);
3550:     } if (EG (exception)) {
3551:         goto done;
3552:     }
3553:
3554:     static int spl_iterator_to_array_apply (zend_object_iterator *iter, void *puser) /* {{{ */
3555:     {
3556:         zval *data, *return_value = (&zval)*puser;
3557:
3558:         data = iter->funcs->get_current_data (iter);
3559:         if (EG (exception)) {
3560:             return ZEND_HASH_APPLY_STOP;
3561:         }
3562:         if (data == NULL) {
3563:             return ZEND_HASH_APPLY_STOP;
3564:         }
3565:         if (iter->funcs->get_current_key (&iter, &key);
3566:             if (EG (exception)) {
3567:                 return ZEND_HASH_APPLY_STOP;
3568:             }
3569:             array_set_zval_key (&ARRVAL_P (return_value), &key, data);
3570:             zval_ptr_dtor (&key);
3571:             else {
3572:                 Z_TRY_ADDREF_P (data);
3573:                 add_next_index_zval (return_value, data);
3574:             }
3575:             return ZEND_HASH_APPLY_KEEP;
3576:         }
3577:     } /* }}} */
3578:
3579: static int spl_iterator_to_values_apply (zend_object_iterator *iter, void *puser) /* {{{ */
3580: {
3581:     zval *data, *return_value = (&zval)*puser;
3582:
3583:     data = iter->funcs->get_current_data (iter);
3584:     if (EG (exception)) {
3585:         return ZEND_HASH_APPLY_STOP;
3586:     }
3587:     if (data == NULL) {
3588:         return ZEND_HASH_APPLY_STOP;
3589:     }
3590:     Z_TRY_ADDREF_P (data);
3591:     add_next_index_zval (return_value, data);
3592:     return ZEND_HASH_APPLY_KEEP;
3593: } /* }}} */
3594:
3595: /* {{{ proto array iterator_to_array(Traversable $t, bool use_keys = true)
3596:  * Copy the iterator into an array */
3597: PHP_FUNCTION (iterator_to_array)
3598: {
3599:     zval *obj;
3600:     zend_bool use_keys = 1;
3601:
3602:     if (zend_parse_parameters (ZEND_NUM_ARGS(), "O|b", &obj, zend_ce_traversable, &use_keys) == FAILURE) {
3603:         return;
3604:     }
3605:     array_init (&return_value);
3606:
3607:     if (spl_iterator_apply (obj, use_keys ? spl_iterator_to_array_apply : spl_iterator_to_values_apply, (&zval)*return_value) != SUCCESS) {
3608:         spl_ptr_dtor (&return_value);
3609:         return NULL();
3610:     }
3611: } /* }}} */
3612:
3613: static int spl_iterator_count_apply (zend_object_iterator *iter, void *puser) /* {{{ */
3614: {
3615:     if (zend_long *puser)++;
3616:     return ZEND_HASH_APPLY_KEEP;
3617: } /* }}} */
3618:
3619: /* {{{ proto int iterator_count(Traversable $t)
3620:  * Count the elements in an iterator */
3621: PHP_FUNCTION (iterator_count)
3622: {
3623:     zval *obj;
3624:     zend_long count = 0;
```

```
3738: REGISTER_SPL_STD_CLASS_EX(IteratorIterator, spl_dual_it_new, spl_funcns_IteratorIterator);
3739: REGISTER_SPL_ITERATOR(IteratorIterator);
3740: REGISTER_SPL_ITERATOR(IteratorIterator);
3741: REGISTER_SPL_IMPLMENTS(IteratorIterator, OuterIterator);
3742:
3743: REGISTER_SPL_SUB_CLASS_EX(FilterIterator, IteratorIterator, spl_dual_it_new, spl_funcns_FilterIterator);
3744: spl_on_FilterIterator->on_flags |= ZEND_ACC_EXPLICIT_ABSTRACT_CLASS;
3745:
3746: REGISTER_SPL_SUB_CLASS_EX(RecursiveFilterIterator, FilterIterator, spl_dual_it_new, spl_funcns_RecursiveFilterIterator);
3747: REGISTER_SPL_IMPLMENTS(RecursiveFilterIterator, RecursiveIterator);
3748:
3749: REGISTER_SPL_SUB_CLASS_EX(CallbackFilterIterator, FilterIterator, spl_dual_it_new, spl_funcns_CallbackFilterIterator);
3750:
3751: REGISTER_SPL_SUB_CLASS_EX(RecursiveCallbackFilterIterator, CallbackFilterIterator, spl_dual_it_new, spl_funcns_RecursiveCallbackFilterIterator);
3752: REGISTER_SPL_IMPLMENTS(RecursiveCallbackFilterIterator, RecursiveIterator);
3753:
3754:
3755: REGISTER_SPL_SUB_CLASS_EX(ParentIterator, RecursiveFilterIterator, spl_dual_it_new, spl_funcns_ParentIterator);
3756:
3757: REGISTER_SPL_INTERFACE(SeekableIterator);
3758: REGISTER_SPL_ITERATOR(SeekableIterator);
3759:
3760: REGISTER_SPL_SUB_CLASS_EX(LimitIterator, IteratorIterator, spl_dual_it_new, spl_funcns_LimitIterator);
3761:
3762: REGISTER_SPL_SUB_CLASS_EX(CachingIterator, IteratorIterator, spl_dual_it_new, spl_funcns_CachingIterator);
3763: REGISTER_SPL_IMPLMENTS(CachingIterator, ArrayAccess);
3764: REGISTER_SPL_IMPLMENTS(CachingIterator, Countable);
3765:
3766: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "CALL_TOSTRING", CIT_CALL_TOSTRING);
3767: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "CATCH_GET_CHILD", CIT_CATCH_GET_CHILD);
3768: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_KEY", CIT_TOSTRING_USE_KEY);
3769: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_CURRENT", CIT_TOSTRING_USE_CURRENT);
3770: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "TOSTRING_USE_INNER", CIT_TOSTRING_USE_INNER);
3771: REGISTER_SPL_CLASS_CONST_LONG(CachingIterator, "FULL_CACHE", CIT_FULL_CACHE);
3772:
3773: REGISTER_SPL_SUB_CLASS_EX(RecursiveCachingIterator, CachingIterator, spl_dual_it_new, spl_funcns_RecursiveCachingIterator);
3774: REGISTER_SPL_IMPLMENTS(RecursiveCachingIterator, RecursiveIterator);
3775:
3776: REGISTER_SPL_SUB_CLASS_EX(NoRewindIterator, IteratorIterator, spl_dual_it_new, spl_funcns_NoRewindIterator);
3777:
3778: REGISTER_SPL_SUB_CLASS_EX(AppendIterator, IteratorIterator, spl_dual_it_new, spl_funcns_AppendIterator);
3779:
3780: REGISTER_SPL_IMPLMENTS(RecursiveIteratorIterator, OuterIterator);
3781:
3782: REGISTER_SPL_SUB_CLASS_EX(InfinitelIterator, IteratorIterator, spl_dual_it_new, spl_funcns_InfinitelIterator);
3783:
3784: #if HAVE_PCRE || HAVE_REGEX_PCRE
3785: REGISTER_SPL_SUB_CLASS_EX(RegexIterator, FilterIterator, spl_dual_it_new, spl_funcns_RegexIterator);
3786: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "USE_KEY", REGIT_USE_KEY);
3787: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "INVERT_MATCH", REGIT_INVERTED);
3788: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "MATCH", REGIT_MATCH_MATCH);
3789: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "GET_MATCH", REGIT_MATCH_GET_MATCH);
3790: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "ALL_MATCHES", REGIT_MATCH_ALL_MATCHES);
3791: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "SPLIT", REGIT_MATCH_SPLIT);
3792: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "REPLACE", REGIT_MATCH_REPLACE);
3793: REGISTER_SPL_CLASS_CONST_LONG(RegexIterator, "PROPERTY", REGIT_MATCH_PROPERTY);
3794: REGISTER_SPL_IMPLMENTS(RecursiveRegexIterator, RecursiveIterator);
3795:
3796: spl_on_RegexIterator = NULL;
3797: spl_on_RecursiveRegexIterator = NULL;
3798: #endif
3799:
3800: REGISTER_SPL_STD_CLASS_EX(EmptyIterator, NULL, spl_funcns_EmptyIterator);
3801: REGISTER_SPL_ITERATOR(EmptyIterator);
3802:
3803: REGISTER_SPL_SUB_CLASS_EX(RecursiveTreeIterator, RecursiveIteratorIterator, spl_RecursiveTreeIterator_new, spl_funcns_RecursiveTreeIterator);
3804: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "BYPASS_CURRENT", RTIT_BYPASS_CURRENT);
3805: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "BYPASS_KEY", RTIT_BYPASS_KEY);
3806: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_LEFT", 0);
3807: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_MID_HAS_NEXT", 1);
3808: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_MID_LAST", 2);
3809: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_END_HAS_NEXT", 3);
3810: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_END_LAST", 4);
3811: REGISTER_SPL_CLASS_CONST_LONG(RecursiveTreeIterator, "PREFIX_RIGHT", 5);
3812:
3813: return SUCCESS;
3814: }
3815: /* }}} */
3816:
3817: /*
3818:  * Local variables:
3819:  * tab-width: 4
3820:  * c-basic-offset: 4
3821:  * End:
3822:  * vim600: fdm=marker
3823:  * vim: noet sw=4 ts=4
3824:  */
```

```
1: /*
2:  *-----*
3:  * | PHP Version 7 |
4:  *-----*
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  *-----*
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  *-----*
15:  * | Authors: Etienne Kneuss <colder@php.net> |
16:  *-----*
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_DLLIST_H
22: #define SPL_DLLIST_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_SplDoublyLinkedList;
28: extern PHPAPI zend_class_entry *spl_ce_SplQueue;
29: extern PHPAPI zend_class_entry *spl_ce_SplStack;
30:
31: PHP_MINIT_FUNCTION(spl_dllist);
32:
33: #endif /* SPL_DLLIST_H */
34:
35: /*
36:  * Local Variables:
37:  * n-basis-offset: 4
38:  * tab-width: 4
39:  * End:
40:  * vim600: fdm=marker
41:  * vim: noet sw=4 ts=4
42:  */
```

```
1: /*
2:  * =====
3:  * | PHP Version ? |
4:  * =====
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * =====
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * =====
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * =====
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef PHP_FUNCTIONS_H
22: #define PHP_FUNCTIONS_H
23:
24: #include "php.h"
25:
26: typedef zend_object* (*create_object_func_t)(zend_class_entry *class_type);
27:
28: #define REGISTER_SPL_STO_CLASS(class_name, obj_ctor) \
29:     spl_register_std_class(spl_ce_ ## class_name, # class_name, obj_ctor, NULL);
30:
31: #define REGISTER_SPL_STO_CLASS_EX(class_name, obj_ctor, funcns) \
32:     spl_register_std_class(spl_ce_ ## class_name, # class_name, obj_ctor, funcns);
33:
34: #define REGISTER_SPL_SUB_CLASS_EX(class_name, parent_class_name, obj_ctor, funcns) \
35:     spl_register_sub_class(spl_ce_ ## class_name, spl_ce_ ## parent_class_name, # class_name, obj_ctor, funcns);
36:
37: #define REGISTER_SPL_INTERFACE(class_name) \
38:     spl_register_interface(spl_ce_ ## class_name, # class_name, spl_funcns_ ## class_name);
39:
40: #define REGISTER_SPL_IMPLEMENTNS(class_name, interface_name) \
41:     zend_class_implements(spl_ce_ ## class_name, 1, spl_ce_ ## interface_name);
42:
43: #define REGISTER_SPL_ITERATOR(class_name) \
44:     zend_class_implements(spl_ce_ ## class_name, 1, zend_ce_iterator);
45:
46: #define REGISTER_SPL_PROPERTY(class_name, prop_name, prop_flags) \
47:     spl_register_property(spl_ce_ ## class_name, prop_name, sizeof(prop_name)-1, prop_flags);
48:
49: #define REGISTER_SPL_CLASS_CONST_LONG(class_name, const_name, value) \
50:     zend_declare_class_constant_long(spl_ce_ ## class_name, const_name, sizeof(const_name)-1, (zend_long)value);
51:
52: void spl_register_std_class(zend_class_entry ** pce, char * class_name, create_object_func_t ctor, const zend_function_entry * function_list);
53: void spl_register_sub_class(zend_class_entry ** pce, zend_class_entry * parent_ce, char * class_name, create_object_func_t ctor, const zend_function_
ntry * function_list);
54: void spl_register_interface(zend_class_entry ** pce, char * class_name, const zend_function_entry * functions);
55:
56: void spl_register_property(zend_class_entry * class_entry, char *prop_name, int prop_name_len, int prop_flags);
57:
58: /* sub: whether to allow subclasses/interfaces
59:    allow = 0: allow all classes and interfaces
60:    allow > 0: allow all that match and mask ce_flags
61:    allow < 0: disallow all that match and mask ce_flags
62: */
63: void spl_add_class_name(zval * list, zend_class_entry * pce, int allow, int ce_flags);
64: void spl_add_interfaces(zval * list, zend_class_entry * pce, int allow, int ce_flags);
65: void spl_add_traits(zval * list, zend_class_entry * pce, int allow, int ce_flags);
66: int spl_add_classes(zend_class_entry *pce, zval *list, int sub, int allow, int ce_flags);
67:
68: /* caller must free(return) */
69: zend_string *spl_get_private_prop_name(zend_class_entry *ce, char *prop_name, int prop_len);
70:
71: #define SPL_ME(class_name, function_name, arg_info, flags) \
72:     PHP_ME(spl, ## class_name, function_name, arg_info, flags)
73:
74: #define SPL_ABSTRACT_ME(class_name, function_name, arg_info) \
75:     ZEND_ABSTRACT_ME(spl, ## class_name, function_name, arg_info)
76:
77: #define SPL_METHOD(class_name, function_name) \
78:     PHP_METHOD(spl, ## class_name, function_name)
79:
80: #define SPL_MA(class_name, function_name, alias_class, alias_function, arg_info, flags) \
81:     PHP_MA(spl, ## alias_class, function_name, alias_function, arg_info, flags)
82: #endif /* PHP_FUNCTIONS_H */
83:
84: /*
85:  * Local Variables:
86:  * c-basic-offset: 4
87:  * tab-width: 4
88:  * End:
89:  * vim600: fdm=marker
90:  * vim: noet sw=4 ts=4
91:  */
```

[illegible]

```
377: if (alif->func_ptr &&
378: UNEXPECTED(alif->func_ptr->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
379: zend_string_release(alif->func_ptr->common.function_name);
380: zend_free_trampoline(alif->func_ptr);
381: }
382: if (!IS_UNDEF(alif->closure)) {
383: zval_ptr_dtor(alif->closure);
384: }
385: efree(alif);
386: }
387:
388: /* {{{ proto void spl_autoload_call(string class_name)
389: Try all registered autoload function to load the requested class */
390: PHP_FUNCTION(spl_autoload_call)
391: {
392: zval *class_name, retval;
393: zend_string *lc_name, *func_name;
394: autoload_func_info *alif;
395:
396: if (zend_parse_parameters(ZEND_NUM_ARGS() | "s", &class_name) == FAILURE || Z_TYPE_P(class_name) != IS_STRING) {
397: return;
398: }
399:
400: if (SP1_G(autoload_functions)) {
401: hashPosition pos;
402: zend_ulong num_idx;
403: zend_function *func;
404: zend_fcall_info fci;
405: zend_fcall_info_cache fci_c;
406: zend_class_entry *called_scope = zend_get_called_scope(execute_data);
407: int l_autoload_running = SP1_G(autoload_running);
408:
409: SP1_G(autoload_running) = 1;
410: lc_name = zend_string_tolower(Z_STR_P(class_name));
411:
412: fci.size = sizeof(fci);
413: fci.retval = &retval;
414: fci.param_count = 1;
415: fci.params = class_name;
416: fci.no_separation = 1;
417:
418: ZVAL_UNDEF(&fci.function_name); /* Unused */
419:
420: zend_hash_internal_pointer_reset_ex(SP1_G(autoload_functions), &pos);
421: while (zend_hash_get_current_key_ex(SP1_G(autoload_functions), &func_name, &num_idx, &pos) == HASH_KEY_IS_STRING) {
422: alif = zend_hash_get_current_data_ptr_ex(SP1_G(autoload_functions), &pos);
423: func = alif->func_ptr;
424: if (UNEXPECTED(func->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
425: func = &alif->closure->zend_op_array;
426: memcpy(func, alif->func_ptr, sizeof(zend_op_array));
427: zend_string_addref(func->op_array.function_name);
428: }
429: ZVAL_UNDEF(&ret_val);
430: fci.func.handler = func;
431: if (!IS_UNDEF(alif->obj)) {
432: fci.object = NULL;
433: fci.object = NULL;
434: fci.calling_scope = alif->obj;
435: if (alif->obj &&
436: (called_scope ||
437: !instanceof_function(called_scope, alif->obj))) {
438: fci.called_scope = alif->obj;
439: } else {
440: fci.called_scope = called_scope;
441: }
442: } else {
443: fci.object = Z_OBJ(alif->obj);
444: fci.object = Z_OBJ(alif->obj);
445: fci.called_scope = Z_OBJCE(alif->obj);
446: }
447:
448: zend_call_function(&fci, &fci_c);
449: zval_ptr_dtor(&ret_val);
450:
451: if (EG(exception)) {
452: break;
453: }
454:
455: if (pos + 1 == SP1_G(autoload_functions)->nNumOfElements) {
456: zend_hash_exists(&EG(class_table), lc_name);
457: break;
458: }
459: zend_hash_move_forward_ex(SP1_G(autoload_functions), &pos);
460: }
461: zend_string_release(lc_name);
462: SP1_G(autoload_running) = l_autoload_running;
463: } else {
464: /* do not use or overwrite EG(autoload_func) here */
465: zend_call_method_with_1_params(NULL, NULL, NULL, "spl_autoload", NULL, class_name);
466: }
467: /* }}} */
468:
469: #define HT_MOVE_TAIL_TO_HEAD(ht) \
470: do { \
471: Bucket tmp = (ht)->data[(ht)->nNumUsed-1]; \
472: memmove((ht)->data + 1, (ht)->data, \
473: sizeof(Bucket) * ((ht)->nNumUsed - 1)); \
474: (ht)->data[0] = tmp; \
475: zend_hash_rehash(ht); \
476: } while (0)
477:
478: /* {{{ proto bool spl_autoload_register([mixed autoload_function [, bool throw [, bool prepend]])
479: Register given function as __autoload() implementation */
480: PHP_FUNCTION(spl_autoload_register)
481: {
482: zend_string *func_name;
483: char *error = NULL;
484: zend_string *lc_name;
485: zval *callable = NULL;
486: zend_bool do_throw = 1;
487: zend_bool prepend = 0;
488: zend_function *spl_func_ptr;
489: autoload_func_info alif;
490: zend_object *obj_ptr;
491: zend_fcall_info_cache fci;
492:
493: if (zend_parse_parameters_ex(ZEND_PARSE_PARAMS_QUIET, ZEND_NUM_ARGS() | "sb", &callable, &do_throw, &prepend) == FAILURE) {
494: return;
495: }
496:
497: if (ZEND_NUM_ARGS() > 0) {
498: if (!is_callable_ex(callable, NULL, IS_CALLABLE_STRICT, &func_name, &fci, &error)) {
499: alif.obj = fci.calling_scope;
500: alif.func_ptr = fci.function_handler;
501: obj_ptr = fci.object;
502: if (Z_TYPE_P(callable) == IS_ARRAY) {
503: if ((obj_ptr && alif.func_ptr && !(alif.func_ptr->common.fn_flags & ZEND_ACC_STATIC)) {
504: if (do_throw) {
505: zend_throw_exception_ex(spl_ce_LogicException, 0, "Passed array specifies a non static method but no object (\"%s\")", error);
506: }
507: if (error) {
508: efree(error);
509: }
510: zend_string_release(func_name);
511: RETURN_FALSE;
512: } else if (do_throw) {
513: zend_throw_exception_ex(spl_ce_LogicException, 0, "Passed array does not specify to method (\"%s\")", alif.func_ptr ? "a callable" : "an exist
ng", obj_ptr ? "static" : "", error);
514: }
515: if (error) {
516: efree(error);
517: }
518: zend_string_release(func_name);
519: RETURN_FALSE;
520: } else if (!Z_TYPE_P(callable) == IS_STRING) {
521: if (do_throw) {
522: zend_throw_exception_ex(spl_ce_LogicException, 0, "Function '%s' not to (\"%s\")", ZSTR_VAL(func_name), alif.func_ptr ? "callable" : "found", err
or);
523: }
524: if (error) {
525: efree(error);
526: }
527: zend_string_release(func_name);
528: RETURN_FALSE;
529: } else {
530: if (do_throw) {
531: zend_throw_exception_ex(spl_ce_LogicException, 0, "Illegal value passed (\"%s\")", error);
532: }
533: if (error) {
534: efree(error);
535: }
536: zend_string_release(func_name);
537: RETURN_FALSE;
538: }
539: } else if (fci.function_handler->type == ZEND_INTERNAL_FUNCTION &&
540: fci.function_handler->internal_function_handler == zif_spl_autoload_call) {
541: if (do_throw) {
542: zend_throw_exception_ex(spl_ce_LogicException, 0, "Function spl_autoload_call() cannot be registered");
543: }
544: if (error) {
545: efree(error);
546: }
547: zend_string_release(func_name);
548: RETURN_FALSE;
549: }
550: alif.obj = fci.calling_scope;
551: alif.func_ptr = fci.function_handler;
552: obj_ptr = fci.object;
553: if (error) {
554: efree(error);
555: }
556:
557: if (Z_TYPE_P(callable) == IS_OBJECT) {
558: ZVAL_COPY(&alif.closure, callable);
559:
560: lc_name = zend_string_alloc(ZSTR_LEN(func_name) + sizeof(uint32_t), 0);
561: zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name));
562: memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(func_name), &Z_OBJ_HANDLE_P(callable), sizeof(uint32_t));
```

```
563: ZSTR_VAL(lc_name) (ZSTR_LEN(lc_name) = '0');
564: } else {
565: ZVAL_UNDEF(&alif.closure);
566: /* Skip loading */
567: if (ZSTR_VAL(func_name)[0] == '\\' {
568: lc_name = zend_string_alloc(ZSTR_LEN(func_name) - 1, 0);
569: zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name) + 1, ZSTR_LEN(func_name) - 1);
570: } else {
571: lc_name = zend_string_tolower(func_name);
572: }
573: }
574: zend_string_release(func_name);
575:
576: if (SP1_G(autoload_functions) && zend_hash_exists(SP1_G(autoload_functions), lc_name)) {
577: if (!IS_UNDEF(alif.closure)) {
578: Z_RELEAF_P(alif.closure);
579: }
580: goto skip;
581: }
582:
583: if (obj_ptr && (alif.func_ptr->common.fn_flags & ZEND_ACC_STATIC)) {
584: /* add obj_ptr to hash to ensure uniqueness, for more reference look at bug #40091 */
585: lc_name = zend_string_extend(lc_name, ZSTR_LEN(lc_name) + sizeof(uint32_t), 0);
586: memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(lc_name) - sizeof(uint32_t), obj_ptr->handle, sizeof(uint32_t));
587: ZSTR_VAL(lc_name) (ZSTR_LEN(lc_name) = '0');
588: ZVAL_OBJ(&alif.obj, obj_ptr);
589: Z_ADDREF(alif.obj);
590: } else {
591: ZVAL_UNDEF(&alif.obj);
592: }
593:
594: if (SP1_G(autoload_functions)) {
595: ALLOC_HASHTABLE(SP1_G(autoload_functions));
596: zend_hash_init(SP1_G(autoload_functions), 1, NULL, autoload_func_info_dtor, 0);
597: }
598:
599: spl_func_ptr = zend_hash_str_find_ptr(EG(function_table), "spl_autoload", sizeof("spl_autoload") - 1);
600:
601: if (EG(autoload_func) == spl_func_ptr) { /* registered already, so we insert that first */
602: autoload_func_info spl_alif;
603:
604: spl_alif.func_ptr = spl_func_ptr;
605: ZVAL_UNDEF(&spl_alif.obj);
606: ZVAL_UNDEF(&spl_alif.closure);
607: spl_alif.obj = NULL;
608: zend_hash_ptr_add_ptr(SP1_G(autoload_functions), "spl_autoload", sizeof("spl_autoload") - 1,
&spl_alif, sizeof(autoload_func_info));
609: if (prepend && SP1_G(autoload_functions)->nNumOfElements > 1) {
610: /* Move the newly created element to the head of the hashtable */
611: HT_MOVE_TAIL_TO_HEAD(SP1_G(autoload_functions));
612: }
613: }
614:
615: if (UNEXPECTED(alif.func_ptr == &EG(trampoline))) {
616: zend_function *copy = &alif->closure->zend_op_array;
617:
618: memcpy(copy, alif.func_ptr, sizeof(zend_op_array));
619: alif.func_ptr->common.function_name = NULL;
620: alif.func_ptr = copy;
621:
622: }
623: if (zend_hash_str_exists(SP1_G(autoload_functions), lc_name, alif, sizeof(autoload_func_info)) == NULL) {
624: if (obj_ptr && (alif.func_ptr->common.fn_flags & ZEND_ACC_STATIC)) {
625: Z_RELEAF_P(alif.obj);
626: }
627: if (!IS_UNDEF(alif.closure)) {
628: Z_RELEAF_P(alif.closure);
629: }
630: if (UNEXPECTED(alif.func_ptr->common.fn_flags & ZEND_ACC_CALL_VIA_TRAMPOLINE)) {
631: zend_string_release(alif.func_ptr->common.function_name);
632: zend_free_trampoline(alif.func_ptr);
633: }
634: }
635: if (prepend && SP1_G(autoload_functions)->nNumOfElements > 1) {
636: /* Move the newly created element to the head of the hashtable */
637: HT_MOVE_TAIL_TO_HEAD(SP1_G(autoload_functions));
638: }
639: skip:
640: zend_string_release(lc_name);
641:
642: if (SP1_G(autoload_functions)) {
643: EG(autoload_func) = zend_hash_str_find_ptr(EG(function_table), "spl_autoload_call", sizeof("spl_autoload_call") - 1);
644: } else {
645: EG(autoload_func) = zend_hash_str_find_ptr(EG(function_table), "spl_autoload", sizeof("spl_autoload") - 1);
646: }
647:
648: RETURN_TRUE;
649: }
650: /* }}} */
651:
652: /* {{{ proto bool spl_autoload_unregister(mixed autoload_function)
653: Unregister given function as __autoload() implementation */
654: PHP_FUNCTION(spl_autoload_unregister)
655: {
656: zend_string *func_name = NULL;
657: char *error = NULL;
658: zend_string *lc_name;
659: zval *callable;
660: int success = FAILURE;
661: zend_function *spl_func_ptr;
662: zend_object *obj_ptr;
663: zend_fcall_info_cache fci;
664:
665: if (zend_parse_parameters(ZEND_NUM_ARGS() | "s", &callable) == FAILURE) {
666: return;
667: }
668:
669: if (!is_callable_ex(callable, NULL, IS_CALLABLE_CHECK_SYNTAX_ONLY, &func_name, &fci, &error)) {
670: zend_throw_exception_ex(spl_ce_LogicException, 0, "Unable to unregister invalid function (\"%s\")", error);
671: if (error) {
672: efree(error);
673: }
674: if (func_name) {
675: zend_string_release(func_name);
676: }
677: RETURN_FALSE;
678: }
679: obj_ptr = fci.object;
680: if (error) {
681: efree(error);
682: }
683:
684: if (Z_TYPE_P(callable) == IS_OBJECT) {
685: lc_name = zend_string_alloc(ZSTR_LEN(func_name) + sizeof(uint32_t), 0);
686: zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name), ZSTR_LEN(func_name));
687: memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(func_name), &Z_OBJ_HANDLE_P(callable), sizeof(uint32_t));
688: ZSTR_VAL(lc_name) (ZSTR_LEN(lc_name) = '0');
689: } else {
690: /* Skip loading */
691: if (ZSTR_VAL(func_name)[0] == '\\' {
692: lc_name = zend_string_alloc(ZSTR_LEN(func_name) - 1, 0);
693: zend_str_tolower_copy(ZSTR_VAL(lc_name), ZSTR_VAL(func_name) + 1, ZSTR_LEN(func_name) - 1);
694: } else {
695: lc_name = zend_string_tolower(func_name);
696: }
697: }
698: zend_string_release(func_name);
699:
700: if (SP1_G(autoload_functions)) {
701: if (ZSTR_LEN(lc_name) == sizeof("spl_autoload_call") - 1 && !strcmp(ZSTR_VAL(lc_name), "spl_autoload_call")) {
702: /* remove call */
703: if (SP1_G(autoload_running)) {
704: zend_hash_destroy(SP1_G(autoload_functions));
705: FREE_HASHTABLE(SP1_G(autoload_functions));
706: SP1_G(autoload_functions) = NULL;
707: EG(autoload_func) = NULL;
708: } else {
709: zend_hash_clean(SP1_G(autoload_functions));
710: }
711: success = SUCCESS;
712: } else {
713: /* remove specific */
714: success = zend_hash_del(SP1_G(autoload_functions), lc_name);
715: if (success != SUCCESS && obj_ptr) {
716: lc_name = zend_string_extend(lc_name, ZSTR_LEN(lc_name) + sizeof(uint32_t), 0);
717: memcpy(ZSTR_VAL(lc_name) + ZSTR_LEN(lc_name) - sizeof(uint32_t), obj_ptr->handle, sizeof(uint32_t));
718: ZSTR_VAL(lc_name) (ZSTR_LEN(lc_name) = '0');
719: success = zend_hash_del(SP1_G(autoload_functions), lc_name);
720: }
721: }
722: }
723: if (ZSTR_LEN(lc_name) == sizeof("spl_autoload")-1 && !strcmp(ZSTR_VAL(lc_name), "spl_autoload")) {
724: /* register single spl_autoload() */
725: spl_func_ptr = zend_hash_str_find_ptr(EG(function_table), "spl_autoload", sizeof("spl_autoload") - 1);
726: if (EG(autoload_func) == spl_func_ptr) {
727: success = SUCCESS;
728: EG(autoload_func) = NULL;
729: }
730: }
731:
732: zend_string_release(lc_name);
733: RETURN_BOOL(success == SUCCESS);
734: /* }}} */
735:
736: /* {{{ proto false|array spl_autoload_functions()
737: Return all registered __autoload() functions */
738: PHP_FUNCTION(spl_autoload_functions)
739: {
740: zend_function *fptr;
741: autoload_func_info *alif;
742:
743: if (zend_parse_parameters_none() == FAILURE) {
744: return;
745: }
746:
747: if (EG(autoload_func)) {
748: if (fptr = zend_hash_str_find_ptr(EG(function_table), ZEND_AUTOLOAD_FUNC_NAME, sizeof(ZEND_AUTOLOAD_FUNC_NAME) - 1)) {
749: array_init(&alif);
750: add_assoc_index_string(return_value, ZEND_AUTOLOAD_FUNC_NAME, sizeof(ZEND_AUTOLOAD_FUNC_NAME)-1);
```

```
751:     return;
752: }
753: RETURN_FALSE;
754: }
755:
756: fptr = zend_hash_str_find_ptr(EG(function_table), "spl_autoload_call", sizeof("spl_autoload_call") - 1);
757:
758: if (EG(autoload_func) == fptr) {
759:     zend_string *key;
760:     array_init(&return_value);
761:     ZEND_HASH_FOREACH_STR_KEY_PTR(SPL_G(autoload_functions), key, ainfo) {
762:         if (!IS_BOOLREF(ainfo->closure)) {
763:             Z_ADDREF(ainfo->closure);
764:             add_next_index_val(&return_value, &ainfo->closure);
765:         } else if (ainfo->func_ptr->common.scope) {
766:             zval tmp;
767:
768:             array_init(&tmp);
769:             if (!IS_BOOLREF(ainfo->obj)) {
770:                 Z_ADDREF(ainfo->obj);
771:                 add_next_index_val(&tmp, &ainfo->obj);
772:             } else {
773:                 add_next_index_str(&tmp, zend_string_copy(ainfo->ce->name));
774:             }
775:             add_next_index_str(&tmp, zend_string_copy(ainfo->func_ptr->common.function_name));
776:             add_next_index_val(&return_value, &tmp);
777:         } else {
778:             if (IS_BOOLREF(EG(autoload_func)->func_ptr->common.function_name), "_lambda_func", &isdef("_lambda_func") - 1) {
779:                 add_next_index_str(&return_value, zend_string_copy(ainfo->func_ptr->common.function_name));
780:             } else {
781:                 add_next_index_str(&return_value, zend_string_copy(key));
782:             }
783:         }
784:     } ZEND_HASH_FOREACH_END();
785:     return;
786: }
787:
788: array_init(&return_value);
789: add_next_index_str(&return_value, zend_string_copy(EG(autoload_func)->common.function_name));
790: /* }}} */
791:
792: /* {{{ proto string spl_object_hash(object obj)
793:  * Returns hash id for given object */
794: PHP_FUNCTION(spl_object_hash)
795: {
796:     zval *obj;
797:
798:     if (zend_parse_parameters(ZEND_NUM_ARGS() & "o", &obj) == FAILURE) {
799:         return;
800:     }
801:
802:     RETURN_NEW_STR(PHP_SPL_OBJECT_HASH(obj));
803: }
804: /* }}} */
805:
806: /* {{{ proto int spl_object_id(object obj)
807:  * Returns the integer object handle for the given object */
808: PHP_FUNCTION(spl_object_id)
809: {
810:     zval *obj;
811:
812:     ZEND_PARSE_PARAMETERS_START(1, 1)
813:     Z_PARAM_OBJ(obj)
814:     ZEND_PARSE_PARAMETERS_END();
815:
816:     RETURN_LONG((zend_long)Z_OBJ_HANDLE_P(obj));
817: }
818: /* }}} */
819:
820: PHPAPI zend_string *php_spl_object_hash(zval *obj) /* {{{ */
821: {
822:     intptr_t hash_handle, hash_handlers;
823:
824:     if (!SPL_G(hash_mask_init)) {
825:         SPL_G(hash_mask_handle) = (intptr_t)(php_mt_rand() >> 1);
826:         SPL_G(hash_mask_handlers) = (intptr_t)(php_mt_rand() >> 1);
827:         SPL_G(hash_mask_init) = 1;
828:     }
829:
830:     hash_handle = SPL_G(hash_mask_handle) * (intptr_t)Z_OBJ_HANDLE_P(obj);
831:     hash_handlers = SPL_G(hash_mask_handlers);
832:
833:     return strprintf(32, "%06x%06x", hash_handle, hash_handlers);
834: }
835: /* }}} */
836:
837: int spl_build_class_list_string(zval *entry, char **list) /* {{{ */
838: {
839:     char *res;
840:
841:     sprintf(res, 0, "%s", *list, Z_STWAL_P(entry));
842:     zfree(*list);
843:     *list = res;
844:     return ZEND_HASH_APPLY_KEEP;
845: } /* }}} */
846:
847: /* {{{ PHP_MININFO(spl)
848:  */
849: PHP_MININFO_FUNCTION(spl)
850: {
851:     zval list;
852:     char *str;
853:
854:     php_info_print_table_start();
855:     php_info_print_table_header(2, "SPL support", "enabled");
856:
857:     array_init(&list);
858:     SPL_LIST_CLASSES(&list, 0, 1, ZEND_ACC_INTERFACE);
859:     str = estrdup("");
860:     zend_hash_apply_with_argument(Z_ARRVAL_P(&list), (apply_func_arg_t)spl_build_class_list_string, &str);
861:     zval_dtor(&list);
862:     php_info_print_table_row(2, "Interfaces", str + 2);
863:     zfree(str);
864:
865:     array_init(&list);
866:     SPL_LIST_CLASSES(&list, 0, -1, ZEND_ACC_INTERFACE);
867:     str = estrdup("");
868:     zend_hash_apply_with_argument(Z_ARRVAL_P(&list), (apply_func_arg_t)spl_build_class_list_string, &str);
869:     zval_dtor(&list);
870:     php_info_print_table_row(2, "Classes", str + 2);
871:     zfree(str);
872:
873:     php_info_print_table_end();
874: }
875: /* }}} */
876:
877: /* {{{ arginfo */
878: ZEND_BEGIN_ARG_INFO_EX(arginfo_iterator_to_array, 0, 0, 1)
879:     ZEND_ARG_CBR_INFO(0, iterator, Traversable, 0)
880:     ZEND_ARG_INFO(0, use_keys)
881: ZEND_END_ARG_INFO();
882:
883: ZEND_BEGIN_ARG_INFO_EX(arginfo_iterator, 0)
884:     ZEND_ARG_CBR_INFO(0, iterator, Traversable, 0)
885: ZEND_END_ARG_INFO();
886:
887: ZEND_BEGIN_ARG_INFO_EX(arginfo_iterator_apply, 0, 0, 2)
888:     ZEND_ARG_CBR_INFO(0, iterator, Traversable, 0)
889:     ZEND_ARG_INFO(0, function)
890:     ZEND_ARG_ARRAY_INFO(0, args, 1)
891: ZEND_END_ARG_INFO();
892:
893: ZEND_BEGIN_ARG_INFO_EX(arginfo_class_parents, 0, 0, 1)
894:     ZEND_ARG_INFO(0, instance)
895:     ZEND_ARG_INFO(0, autoload)
896: ZEND_END_ARG_INFO();
897:
898: ZEND_BEGIN_ARG_INFO_EX(arginfo_class_implements, 0, 0, 1)
899:     ZEND_ARG_INFO(0, what)
900:     ZEND_ARG_INFO(0, autoload)
901: ZEND_END_ARG_INFO();
902:
903: ZEND_BEGIN_ARG_INFO_EX(arginfo_class_uses, 0, 0, 1)
904:     ZEND_ARG_INFO(0, what)
905:     ZEND_ARG_INFO(0, autoload)
906: ZEND_END_ARG_INFO();
907:
908:
909: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_classes, 0)
910:     ZEND_ARG_INFO(0)
911:
912: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_functions, 0)
913:     ZEND_ARG_INFO(0)
914:
915: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload, 0, 0, 1)
916:     ZEND_ARG_INFO(0, class_name)
917:     ZEND_ARG_INFO(0, file_extensions)
918: ZEND_END_ARG_INFO();
919:
920: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_extensions, 0, 0, 0)
921:     ZEND_ARG_INFO(0, file_extensions)
922: ZEND_END_ARG_INFO();
923:
924: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_call, 0, 0, 1)
925:     ZEND_ARG_INFO(0, class_name)
926: ZEND_END_ARG_INFO();
927:
928: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_register, 0, 0, 0)
929:     ZEND_ARG_INFO(0, autoload_function)
930:     ZEND_ARG_INFO(0, throw)
931:     ZEND_ARG_INFO(0, prepend)
932: ZEND_END_ARG_INFO();
933:
934: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_autoload_unregister, 0, 0, 1)
935:     ZEND_ARG_INFO(0, autoload_function)
936: ZEND_END_ARG_INFO();
937:
938: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_object_hash, 0, 0, 1)
```

```
939:     ZEND_ARG_INFO(0, obj)
940: ZEND_END_ARG_INFO();
941:
942: ZEND_BEGIN_ARG_INFO_EX(arginfo_spl_object_id, 0, 0, 1)
943:     ZEND_ARG_INFO(0, obj)
944: ZEND_END_ARG_INFO();
945: /* }}} */
946:
947: /* {{{ spl_functions
948:  */
949: static const zend_function_entry spl_functions[] = {
950:     PHP_FE(spl_classes, arginfo_spl_classes)
951:     PHP_FE(spl_autoload, arginfo_spl_autoload)
952:     PHP_FE(spl_autoload_extensions, arginfo_spl_autoload_extensions)
953:     PHP_FE(spl_autoload_register, arginfo_spl_autoload_register)
954:     PHP_FE(spl_autoload_unregister, arginfo_spl_autoload_unregister)
955:     PHP_FE(spl_autoload_functions, arginfo_spl_autoload_functions)
956:     PHP_FE(spl_autoload_call, arginfo_spl_autoload_call)
957:     PHP_FE(class_parents, arginfo_class_parents)
958:     PHP_FE(class_implements, arginfo_class_implements)
959:     PHP_FE(class_uses, arginfo_class_uses)
960:     PHP_FE(spl_object_hash, arginfo_spl_object_hash)
961:     PHP_FE(spl_object_id, arginfo_spl_object_id)
962: #ifdef SPL_ITERATORS_H
963:     PHP_FE(iterator_to_array, arginfo_iterator_to_array)
964:     PHP_FE(iterator_count, arginfo_iterator)
965:     PHP_FE(iterator_apply, arginfo_iterator_apply)
966: #endif /* SPL_ITERATORS_H */
967:     PHP_FE_END
968: };
969: /* }}} */
970:
971: /* {{{ PHP_MINIT_FUNCTION(spl)
972:  */
973: PHP_MINIT_FUNCTION(spl)
974: {
975:     PHP_MINIT(spl_exceptions) (INIT_FUNC_ARGS_PASSTHRU);
976:     PHP_MINIT(spl_iterators) (INIT_FUNC_ARGS_PASSTHRU);
977:     PHP_MINIT(spl_array) (INIT_FUNC_ARGS_PASSTHRU);
978:     PHP_MINIT(spl_directory) (INIT_FUNC_ARGS_PASSTHRU);
979:     PHP_MINIT(spl_dlist) (INIT_FUNC_ARGS_PASSTHRU);
980:     PHP_MINIT(spl_heap) (INIT_FUNC_ARGS_PASSTHRU);
981:     PHP_MINIT(spl_directory) (INIT_FUNC_ARGS_PASSTHRU);
982:     PHP_MINIT(spl_observers) (INIT_FUNC_ARGS_PASSTHRU);
983:
984:     return SUCCESS;
985: }
986: /* }}} */
987:
988: PHP_RINIT_FUNCTION(spl) /* {{{ */
989: {
990:     SPL_G(autoload_extensions) = NULL;
991:     SPL_G(autoload_functions) = NULL;
992:     SPL_G(hash_mask_init) = 0;
993:     return SUCCESS;
994: } /* }}} */
995:
996: PHP_SHUTDOWN_FUNCTION(spl) /* {{{ */
997: {
998:     if (SPL_G(autoload_extensions)) {
999:         zend_string_release(SPL_G(autoload_extensions));
1000:         SPL_G(autoload_extensions) = NULL;
1001:     }
1002:     if (SPL_G(autoload_functions)) {
1003:         zend_hash_destroy(SPL_G(autoload_functions));
1004:         FREE_HASHTABLE(SPL_G(autoload_functions));
1005:         SPL_G(autoload_functions) = NULL;
1006:     }
1007:     if (SPL_G(hash_mask_init)) {
1008:         SPL_G(hash_mask_init) = 0;
1009:     }
1010:     return SUCCESS;
1011: } /* }}} */
1012:
1013: /* {{{ spl_module_entry
1014:  */
1015: zend_module_entry spl_module_entry = {
1016:     STANDARD_MODULE_HEADER,
1017:     "spl",
1018:     spl_functions,
1019:     PHP_MINIT(spl),
1020:     NULL,
1021:     PHP_RINIT(spl),
1022:     PHP_SHUTDOWN(spl),
1023:     PHP_MININFO(spl),
1024:     PHP_SPL_VERSION,
1025:     PHP_MODULE_GLOBALS(spl),
1026:     PHP_GINIT(spl),
1027:     NULL,
1028:     NULL,
1029:     STANDARD_MODULE_PROPERTIES_EX
1030: };
1031: /* }}} */
1032:
1033: /*
1034:  * Local variables:
1035:  * tab-width: 4
1036:  * c-basic-offset: 4
1037:  * End:
1038:  * vim600: fdm=marker
1039:  * vim: noet sw=4 ts=4
1040:  */
```



```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifdef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "php.h"
26: #include "php_ini.h"
27: #include "ext/standard/info.h"
28: #include "zend_interfaces.h"
29: #include "zend_exceptions.h"
30:
31: #include "spl_spl.h"
32: #include "spl_functions.h"
33: #include "spl_engine.h"
34: #include "spl_exceptions.h"
35:
36: PHPAPI zend_class_entry *spl_ce_LogicException;
37: PHPAPI zend_class_entry *spl_ce_BadFunctionCallException;
38: PHPAPI zend_class_entry *spl_ce_BadMethodCallException;
39: PHPAPI zend_class_entry *spl_ce_DomainException;
40: PHPAPI zend_class_entry *spl_ce_InvalidArgumentException;
41: PHPAPI zend_class_entry *spl_ce_LengthException;
42: PHPAPI zend_class_entry *spl_ce_OutOfRangeException;
43: PHPAPI zend_class_entry *spl_ce_RuntimeException;
44: PHPAPI zend_class_entry *spl_ce_OutOfBoundsException;
45: PHPAPI zend_class_entry *spl_ce_OverflowException;
46: PHPAPI zend_class_entry *spl_ce_RangeException;
47: PHPAPI zend_class_entry *spl_ce_UnderflowException;
48: PHPAPI zend_class_entry *spl_ce_UnexpectedValueException;
49:
50: #define spl_ce_Exception zend_ce_exception
51:
52: /* {{{ PHP_MINIT_FUNCTION(spl_exceptions) */
53: PHP_MINIT_FUNCTION(spl_exceptions)
54: {
55:     REGISTER_SPL_SUB_CLASS_EX(LogicException, Exception, NULL, NULL);
56:     REGISTER_SPL_SUB_CLASS_EX(BadFunctionCallException, LogicException, NULL, NULL);
57:     REGISTER_SPL_SUB_CLASS_EX(BadMethodCallException, BadFunctionCallException, NULL, NULL);
58:     REGISTER_SPL_SUB_CLASS_EX(DomainException, LogicException, NULL, NULL);
59:     REGISTER_SPL_SUB_CLASS_EX(InvalidArgumentException, LogicException, NULL, NULL);
60:     REGISTER_SPL_SUB_CLASS_EX(LengthException, LogicException, NULL, NULL);
61:     REGISTER_SPL_SUB_CLASS_EX(OutOfRangeException, LogicException, NULL, NULL);
62:
63:     REGISTER_SPL_SUB_CLASS_EX(RuntimeException, Exception, NULL, NULL);
64:     REGISTER_SPL_SUB_CLASS_EX(OutOfBoundsException, RuntimeException, NULL, NULL);
65:     REGISTER_SPL_SUB_CLASS_EX(OverflowException, RuntimeException, NULL, NULL);
66:     REGISTER_SPL_SUB_CLASS_EX(RangeException, RuntimeException, NULL, NULL);
67:     REGISTER_SPL_SUB_CLASS_EX(UnderflowException, RuntimeException, NULL, NULL);
68:     REGISTER_SPL_SUB_CLASS_EX(UnexpectedValueException, RuntimeException, NULL, NULL);
69:
70:     return SUCCESS;
71: }
72: /* }}} */
73:
74: /*
75:  * Local variables:
76:  * tab-width: 4
77:  * c-basic-offset: 4
78:  * End:
79:  * vim600: fdm=marker
80:  * vim: noet sw=4 ts=4
81:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_OBSERVER_H
22: #define SPL_OBSERVER_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_SplObserver;
28: extern PHPAPI zend_class_entry *spl_ce_SplSubject;
29: extern PHPAPI zend_class_entry *spl_ce_SplObjectStorage;
30: extern PHPAPI zend_class_entry *spl_ce_MultiPlaiterator;
31:
32: PHP_MINIT_FUNCTION(spl_observer);
33:
34: #endif /* SPL_OBSERVER_H */
35:
36: /*
37:  * Local Variables:
38:  * c-basic-offset: 4
39:  * tab-width: 4
40:  * End:
41:  * vim600: fdm=marker
42:  * vim: noet sw=4 ts=4
43:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_EXCEPTIONS_H
22: #define SPL_EXCEPTIONS_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_LogicalException;
28: extern PHPAPI zend_class_entry *spl_ce_BadFunctionCallException;
29: extern PHPAPI zend_class_entry *spl_ce_BadMethodCallException;
30: extern PHPAPI zend_class_entry *spl_ce_DomainException;
31: extern PHPAPI zend_class_entry *spl_ce_InvalidArgumentException;
32: extern PHPAPI zend_class_entry *spl_ce_LengthException;
33: extern PHPAPI zend_class_entry *spl_ce_OutOfRangeException;
34:
35: extern PHPAPI zend_class_entry *spl_ce_RuntimeException;
36: extern PHPAPI zend_class_entry *spl_ce_OutOfBoundsException;
37: extern PHPAPI zend_class_entry *spl_ce_OverflowException;
38: extern PHPAPI zend_class_entry *spl_ce_RangeException;
39: extern PHPAPI zend_class_entry *spl_ce_UnderflowException;
40: extern PHPAPI zend_class_entry *spl_ce_UnexpectedValueException;
41:
42: PHP_MINIT_FUNCTION(spl_exceptions);
43:
44: #endif /* SPL_EXCEPTIONS_H */
45:
46: /*
47:  * Local Variables:
48:  * c-basic-offset: 4
49:  * tab-width: 4
50:  * End:
51:  * vim600: fdm=marker
52:  * vim: noet sw=4 ts=4
53:  */
```

```
1: /*
2:  * =====
3:  * | PHP Version 7 |
4:  * =====
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * =====
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * =====
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * =====
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef HAVE_CONFIG_H
22: #include "config.h"
23: #endif
24:
25: #include "php.h"
26: #include "php_ini.h"
27: #include "ext/standard/info.h"
28: #include "php_spl.h"
29:
30: /* {{{ spl_register_interface */
31: void spl_register_interface zend_class_entry ** pce, char * class_name, const zend_function_entry * functions)
32: {
33:     zend_class_entry ce;
34:
35:     INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), functions);
36:     *pce = zend_register_internal_interface(&ce);
37: }
38: /* }}} */
39:
40: /* {{{ spl_register_std_class */
41: PHPAPI void spl_register_std_class(zend_class_entry ** pce, char * class_name, void * obj_ctor, const zend_function_entry * function_list)
42: {
43:     zend_class_entry ce;
44:
45:     INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), function_list);
46:     *pce = zend_register_internal_class(&ce);
47:
48:     /* entries changed by initialize */
49:     if (obj_ctor) {
50:         (*pce->create_object) = obj_ctor;
51:     }
52: }
53: /* }}} */
54:
55: /* {{{ spl_register_sub_class */
56: PHPAPI void spl_register_sub_class(zend_class_entry ** pce, zend_class_entry * parent_ce, char * class_name, void * obj_ctor, const zend_function_entry * function_list)
57: {
58:     zend_class_entry ce;
59:
60:     INIT_CLASS_ENTRY_EX(ce, class_name, strlen(class_name), function_list);
61:     *pce = zend_register_internal_class_ex(&ce, parent_ce);
62:
63:     /* entries changed by initialize */
64:     if (obj_ctor) {
65:         (*pce->create_object) = obj_ctor;
66:     } else {
67:         (*pce->create_object) = parent_ce->create_object;
68:     }
69: }
70: /* }}} */
71:
72: /* {{{ spl_register_property */
73: void spl_register_property(zend_class_entry * class_entry, char *prop_name, int prop_name_len, int prop_flags)
74: {
75:     zend_declare_property_null(class_entry, prop_name, prop_name_len, prop_flags);
76: }
77: /* }}} */
78:
79: /* {{{ spl_add_class_name */
80: void spl_add_class_name(zval *list, zend_class_entry *pce, int allow, int ce_flags)
81: {
82:     if ((allow && (0 < pce->ce_flags & ce_flags)) || (allow && ! (pce->ce_flags & ce_flags))) {
83:         zval *tmp;
84:
85:         if ((tmp = zend_hash_find(Z_ARRVAL_P(list), pce->name)) == NULL) {
86:             zval tmp;
87:             ZVAL_STR_COPY(&tmp, pce->name);
88:             zend_hash_add(Z_ARRVAL_P(list), pce->name, &tmp);
89:         }
90:     }
91: }
92: /* }}} */
93:
94: /* {{{ spl_add_interfaces */
95: void spl_add_interfaces(zval *list, zend_class_entry * pce, int allow, int ce_flags)
96: {
97:     uint32_t num_interfaces;
98:
99:     for (num_interfaces = 0; num_interfaces < pce->num_interfaces; num_interfaces++) {
100:         spl_add_class_name(list, pce->interfaces[num_interfaces], allow, ce_flags);
101:     }
102: }
103: /* }}} */
104:
105: /* {{{ spl_add_traits */
106: void spl_add_traits(zval *list, zend_class_entry * pce, int allow, int ce_flags)
107: {
108:     uint32_t num_traits;
109:
110:     for (num_traits = 0; num_traits < pce->num_traits; num_traits++) {
111:         spl_add_class_name(list, pce->traits[num_traits], allow, ce_flags);
112:     }
113: }
114: /* }}} */
115:
116: /* {{{ spl_add_classes */
117: int spl_add_classes(zend_class_entry *pce, zval *list, int sub, int allow, int ce_flags)
118: {
119:     if (!pce) {
120:         return 0;
121:     }
122:     spl_add_class_name(list, pce, allow, ce_flags);
123:     if (sub) {
124:         spl_add_interfaces(list, pce, allow, ce_flags);
125:         while (pce->parent) {
126:             pce = pce->parent;
127:             spl_add_classes(pce, list, sub, allow, ce_flags);
128:         }
129:     }
130:     return 0;
131: }
132: /* }}} */
133:
134: zend_string * spl_get_private_prop_name(zend_class_entry *ce, char *prop_name, int prop_len) /* {{{ */
135: {
136:     return zend_mangle_property_name(ZSTR_VAL(ce->name), ZSTR_LEN(ce->name), prop_name, prop_len, 0);
137: }
138: /* }}} */
139:
140: /*
141:  * Local Variables:
142:  * tab-width: 4
143:  * c-basic-offset: 4
144:  * End:
145:  * vim600: fdm=marker
146:  * vim: noet sw=4 ts=4
147:  */
148: */
```

```
1: /*
2:  *
3:  * PHP Version 7
4:  *
5:  * Copyright (c) 1997-2018 The PHP Group
6:  *
7:  * This source file is subject to version 3.01 of the PHP license,
8:  * that is bundled with this package in the file LICENSE, and is
9:  * available through the world-wide-web at the following url:
10:  * https://www.php.net/license/3.01.txt
11:  * If you did not receive a copy of the PHP license and are unable to
12:  * obtain it through the world-wide-web, please send a note to
13:  * license@php.net so we can mail you a copy immediately.
14:  *
15:  * Authors: Marcus Boerger <helly@php.net>
16:  *
17:  */
18:
19: /* $Id$ */
20:
21: #ifndef SPL_DIRECTORY_H
22: #define SPL_DIRECTORY_H
23:
24: #include "php.h"
25: #include "php_spl.h"
26:
27: extern PHPAPI zend_class_entry *spl_ce_SplFileInfo;
28: extern PHPAPI zend_class_entry *spl_ce_DirectoryIterator;
29: extern PHPAPI zend_class_entry *spl_ce_FilesystemIterator;
30: extern PHPAPI zend_class_entry *spl_ce_RecursiveDirectoryIterator;
31: extern PHPAPI zend_class_entry *spl_ce_GlobIterator;
32: extern PHPAPI zend_class_entry *spl_ce_SplFileInfo;
33: extern PHPAPI zend_class_entry *spl_ce_SplTempFileObject;
34:
35: PHP_MINIT_FUNCTION(spl_directory);
36:
37: typedef enum {
38:     SPL_FS_INFO, /* must be 0 */
39:     SPL_FS_DIR,
40:     SPL_FS_FILE,
41: } SPL_FS_OBJ_TYPE;
42:
43: typedef struct _spl_filesystem_object spl_filesystem_object;
44:
45: typedef void (*spl_foreign_dtor_t)(spl_filesystem_object *object);
46: typedef void (*spl_foreign_clone_t)(spl_filesystem_object *src, spl_filesystem_object *dst);
47:
48: PHPAPI char* spl_filesystem_object_get_path(spl_filesystem_object *intern, size_t *len);
49:
50: typedef struct _spl_other_handler {
51:     spl_foreign_dtor_t dtor;
52:     spl_foreign_clone_t clone;
53: } spl_other_handler;
54:
55: /* define an overloaded iterator structure */
56: typedef struct {
57:     zend_object_iterator intern;
58:     zval *current;
59:     void *object;
60: } spl_filesystem_iterator;
61:
62: struct _spl_filesystem_object {
63:     void *other;
64:     const spl_other_handler *other_handler;
65:     char *_path;
66:     size_t _path_len;
67:     char *_full_path;
68:     char *_file_name;
69:     size_t file_name_len;
70:     SPL_FS_OBJ_TYPE type;
71:     zend_long flags;
72:     zend_class_entry *file_class;
73:     zend_class_entry *info_class;
74:     union {
75:         struct {
76:             php_stream *stream;
77:             php_stream_dirent entry;
78:             char *sub_path;
79:             size_t sub_path_len;
80:             int index;
81:             int is_recursive;
82:             zend_function *func_read;
83:             zend_function *func_next;
84:             zend_function *func_valid;
85:         } dir;
86:         struct {
87:             php_stream *stream;
88:             php_stream_context *context;
89:             zval *scontext;
90:             char *open_mode;
91:             size_t open_mode_len;
92:             zval current_zval;
93:             char *current_line;
94:             size_t current_line_len;
95:             size_t max_line_len;
96:             zend_long current_line_num;
97:             zval resources;
98:             zend_function *func_getCurr;
99:             char *delimiter;
100:             char *enclosure;
101:             char *escape;
102:         } file;
103:     } u;
104:     zend_object std;
105: };
106:
107: static inline spl_filesystem_object *spl_filesystem_from_obj(zend_object *obj) /* {{{ */ {
108:     return (spl_filesystem_object*)((char*)(obj) - XOffsetOf(spl_filesystem_object, std));
109: }
110: /* }}} */
111:
112: #define Z_SPL_FILESYSTEM_P(zv) spl_filesystem_from_obj(Z_OBJ_P(zv))
113:
114: static inline spl_filesystem_iterator *spl_filesystem_object_to_iterator(spl_filesystem_object *obj)
115: {
116:     spl_filesystem_iterator *it;
117:
118:     it = ecalloc(1, sizeof(spl_filesystem_iterator));
119:     it->object = (void *)obj;
120:     zend_iterator_init(it->intern);
121:     return it;
122: }
123:
124: static inline spl_filesystem_object* spl_filesystem_iterator_to_object(spl_filesystem_iterator *it)
125: {
126:     return (spl_filesystem_object*)it->object;
127: }
128:
129: #define SPL_FILE_OBJECT_DROP_NEW_LINE 0x00000001 /* drop new lines */
130: #define SPL_FILE_OBJECT_READ_HEADER 0x00000002 /* read or read/next */
131: #define SPL_FILE_OBJECT_SKIP_EMPTY 0x00000004 /* skip empty lines */
132: #define SPL_FILE_OBJECT_READ_CSV 0x00000008 /* read via fgets */
133: #define SPL_FILE_OBJECT_MASK 0x0000000F /* read via fgets */
134:
135: #define SPL_FILE_DIR_CURRENT_AS_FILEINFO 0x00000000 /* make RecursiveDirectoryTree::current() return SplFileInfo */
136: #define SPL_FILE_DIR_CURRENT_AS_ZIP 0x00000001 /* make RecursiveDirectoryTree::current() return getZip() */
137: #define SPL_FILE_DIR_CURRENT_AS_PATHNAME 0x00000020 /* make RecursiveDirectoryTree::current() return getPathname() */
138: #define SPL_FILE_DIR_CURRENT_MODE_MASK 0x000000F0 /* mask RecursiveDirectoryTree::current() */
139: #define SPL_FILE_DIR_CURRENT_MODE_MASK ((intern->flags & SPL_FILE_DIR_CURRENT_MODE_MASK) ~ mode)
140:
141: #define SPL_FILE_DIR_KEY_AS_PATHNAME 0x00000000 /* make RecursiveDirectoryTree::key() return getPathname() */
142: #define SPL_FILE_DIR_KEY_AS_PATHNAME 0x00000000 /* make RecursiveDirectoryTree::key() return getPathname() */
143: #define SPL_FILE_DIR_FOLLOW_SYMLINKS 0x00000020 /* make RecursiveDirectoryTree::hasChildren() follow symlinks */
144: #define SPL_FILE_DIR_KEY_MODE_MASK 0x000000F0 /* mask RecursiveDirectoryTree::key() */
145: #define SPL_FILE_DIR_KEY_MODE_MASK ((intern->flags & SPL_FILE_DIR_KEY_MODE_MASK) ~ mode)
146:
147: #define SPL_FILE_DIR_SKIPDOTS 0x00000100 /* Tells whether it should skip dots or not */
148: #define SPL_FILE_DIR_UNIFYPATHS 0x00000200 /* whether to unify path separators */
149: #define SPL_FILE_DIR_OTHERS_MASK 0x00000300 /* mask used for get/setFlags */
150:
151: #endif /* SPL_DIRECTORY_H */
152:
153: /*
154:  * Local Variables:
155:  * c-basic-offset: 4
156:  * tab-width: 4
157:  * End:
158:  * vim600: fdm=marker
159:  * vim: noet sw=4 ts=4
160:  */
```

```
1: /*
2:  * PHP Version 7
3:  *
4:  * Copyright (c) 1997-2018 The PHP Group
5:  *
6:  * This source file is subject to version 3.01 of the PHP license,
7:  * that is bundled with this package in the file LICENSE, and is
8:  * available through the world-wide-web at the following url:
9:  * http://www.php.net/license/3_01.txt
10:  * If you did not receive a copy of the PHP license and are unable to
11:  * obtain it through the world-wide-web, please send a note to
12:  * license@php.net so we can mail you a copy immediately.
13:  *
14:  * Author: Antony Dvayal <trond@daylesday.org>
15:  *
16:  * Elzanne Kneuss <elzanne@php.net>
17:  */
18:
19:
20: /* $Id$ */
21:
22: #ifndef HAVE_CONFIG_H
23: #include "config.h"
24: #endif
25:
26: #include "php.h"
27: #include "php_ini.h"
28: #include "ext/standard/info.h"
29: #include "zend_exceptions.h"
30:
31: #include "spl_spl.h"
32: #include "spl_functions.h"
33: #include "spl_engine.h"
34: #include "spl_fixedarray.h"
35: #include "spl_exceptions.h"
36: #include "spl_iterators.h"
37:
38: zend_object_handlers spl_handler_SplFixedArray;
39: PHP_API zend_class_entry *spl_ce_SplFixedArray;
40:
41: #ifndef COMPILE_DL_SPL_FIXEDARRAY
42: #error COMPILE_DL_SPL_FIXEDARRAY
43: #endif
44:
45: typedef struct _spl_fixedarray { /* {{{ */
46:     zend_long size;
47:     zval *elements;
48:     spl_fixedarray;
49: } /* }}} */
50:
51: typedef struct _spl_fixedarray_object { /* {{{ */
52:     spl_fixedarray array;
53:     zend_function *fptr_offset_get;
54:     zend_function *fptr_offset_set;
55:     zend_function *fptr_offset_has;
56:     zend_function *fptr_offset_del;
57:     zend_function *fptr_count;
58:     spl_ce current;
59:     int flags;
60:     zend_class_entry *ce_iterator;
61:     zend_object std;
62:     spl_fixedarray_object;
63: } /* }}} */
64:
65: typedef struct _spl_fixedarray_it { /* {{{ */
66:     zend_user_iterator intern;
67:     spl_fixedarray_it;
68: } /* }}} */
69:
70: #define SPL_FIXEDARRAY_OVERLOADED_REWIND 0x0001
71: #define SPL_FIXEDARRAY_OVERLOADED_VALID 0x0002
72: #define SPL_FIXEDARRAY_OVERLOADED_KEY 0x0004
73: #define SPL_FIXEDARRAY_OVERLOADED_CURRENT 0x0008
74: #define SPL_FIXEDARRAY_OVERLOADED_NEXT 0x0010
75:
76: static inline spl_fixedarray_object *spl_fixed_array_from_obj(zend_object *obj) /* {{{ */ {
77:     return (spl_fixedarray_object*)((char*)obj) - XFFOFFSET(spl_fixedarray_object, std);
78: } /* }}} */
79:
80: #define SPL_FIXEDARRAY_P(rv) spl_fixed_array_from_obj(Z_OBJ_P(rv))
81:
82: static void spl_fixedarray_init(spl_fixedarray *array, zend_long size) /* {{{ */ {
83:     if (size > 0) {
84:         array->size = 0; /* reset size in case realloc() fails */
85:         array->elements = ecalloc(size, sizeof(zval));
86:         array->size = size;
87:     } else {
88:         array->elements = NULL;
89:         array->size = 0;
90:     }
91: } /* }}} */
92:
93: static void spl_fixedarray_resize(spl_fixedarray *array, zend_long size) /* {{{ */ {
94:     if (size == array->size) {
95:         /* nothing to do */
96:         return;
97:     }
98:     /* first initialization */
99:     if (array->size == 0) {
100:         spl_fixedarray_init(array, size);
101:         return;
102:     }
103:     /* clearing the array */
104:     if (size == 0) {
105:         zend_long i;
106:         for (i = 0; i < array->size; i++) {
107:             zval_ptr_dtor(&(array->elements[i]));
108:         }
109:         array->elements = NULL;
110:     } else if (size > array->size) {
111:         array->elements = safe_realloc(array->elements, size, sizeof(zval), 0);
112:         memset(array->elements + array->size, '\0', sizeof(zval) * (size - array->size));
113:     } else { /* size < array->size */
114:         zend_long i;
115:         for (i = size; i < array->size; i++) {
116:             zval_ptr_dtor(&(array->elements[i]));
117:         }
118:         array->elements = realloc(array->elements, sizeof(zval) * size);
119:     }
120:     array->size = size;
121: } /* }}} */
122:
123: static void spl_fixedarray_copy(spl_fixedarray *to, spl_fixedarray *from) /* {{{ */ {
124:     int i;
125:     for (i = 0; i < from->size; i++) {
126:         ZVAL_COPY(&(to->elements[i]), &(from->elements[i]));
127:     }
128:     to->size = from->size;
129: } /* }}} */
130:
131: static HashTable* spl_fixedarray_object_get_ge(zval *obj, zval **table, int *n) /* {{{ */ {
132:     spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(obj);
133:     HashTable *ht = zend_std_get_properties(obj);
134:     *table = intern->array.elements;
135:     *n = (int)intern->array.size;
136:     return ht;
137: } /* }}} */
138:
139: static HashTable* spl_fixedarray_object_get_properties(zval *obj) /* {{{ */ {
140:     spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(obj);
141:     HashTable *ht = zend_std_get_properties(obj);
142:     zend_long i = 0;
143:     if (intern->array.size > 0) {
144:         zend_long j = zend_hash_num_elements(ht);
145:         for (i = 0; i < intern->array.size; i++) {
146:             if (IS_UNDEF(intern->array.elements[i])) {
147:                 zend_hash_index_update(ht, i, spl_fixedarray_object_get_ge(obj));
148:             } else {
149:                 zend_hash_index_update(ht, i, &SPLFIXEDARRAY_P(obj));
150:             }
151:         }
152:     }
153:     return ht;
154: } /* }}} */
155:
156: static void spl_fixedarray_object_free_storage(zend_object *obj) /* {{{ */ {
157:     spl_fixedarray_object *intern = spl_fixed_array_from_obj(obj);
158:     zend_long i;
```

```
159:     zend_long i;
160:     if (intern->array.size > 0) {
161:         for (i = 0; i < intern->array.size; i++) {
162:             zval_ptr_dtor(&(intern->array.elements[i]));
163:         }
164:     }
165:     if (intern->array.size > 0 & intern->array.elements) {
166:         efree(intern->array.elements);
167:     }
168:     intern->array.size = 0;
169:     zend_object_std_dtor(intern->std);
170: } /* }}} */
171:
172: zend_object_iterator *spl_fixedarray_get_iterator(zend_class_entry *ce, zval *object, int by_ref) {
173:     static zend_object *spl_fixedarray_object_new_ce(zend_class_entry *class_type, zval *orig, int clone_orig) /* {{{ */ {
174:         spl_fixedarray_object *intern;
175:         zend_class_entry *parent = class_type;
176:         int inherited = 0;
177:         intern = zend_object_alloc(sizeof(spl_fixedarray_object), parent);
178:         zend_object_std_init(intern->std, class_type);
179:         object->properties_table(intern->std, class_type);
180:         intern->current = 0;
181:         intern->flags = 0;
182:         if (orig && clone_orig) {
183:             spl_fixedarray_object *other = Z_SPLFIXEDARRAY_P(orig);
184:             intern->ce_get_iterator = other->ce_get_iterator;
185:             spl_fixedarray_init(intern->array, other->array.size);
186:             spl_fixedarray_copy(intern->array, other->array);
187:         }
188:         while (parent) {
189:             if (parent == spl_ce_SplFixedArray) {
190:                 intern->std.handlers = spl_handler_SplFixedArray;
191:                 class_type->get_iterator = spl_fixedarray_get_iterator;
192:                 break;
193:             }
194:             parent = parent->parent;
195:             inherited = 1;
196:         }
197:         if (!parent) { /* this must never happen */
198:             php_error_fatal("Internal compiler error, Class is not child of SplFixedArray");
199:         }
200:         if (class_type->iterator_funcs.if_rewind == zend_hash_str_find_ptr(class_type->function_table, "rewind", sizeof("rewind") - 1)) {
201:             class_type->iterator_funcs.if_valid = zend_hash_str_find_ptr(class_type->function_table, "valid", sizeof("valid") - 1);
202:             class_type->iterator_funcs.if_key = zend_hash_str_find_ptr(class_type->function_table, "key", sizeof("key") - 1);
203:             class_type->iterator_funcs.if_current = zend_hash_str_find_ptr(class_type->function_table, "current", sizeof("current") - 1);
204:             class_type->iterator_funcs.if_next = zend_hash_str_find_ptr(class_type->function_table, "next", sizeof("next") - 1);
205:         }
206:         if (inherited) {
207:             if (class_type->iterator_funcs.if_rewind->common.scope != parent) {
208:                 intern->flags |= SPL_FIXEDARRAY_OVERLOADED_REWIND;
209:             }
210:             if (class_type->iterator_funcs.if_valid->common.scope != parent) {
211:                 intern->flags |= SPL_FIXEDARRAY_OVERLOADED_VALID;
212:             }
213:             if (class_type->iterator_funcs.if_key->common.scope != parent) {
214:                 intern->flags |= SPL_FIXEDARRAY_OVERLOADED_KEY;
215:             }
216:             if (class_type->iterator_funcs.if_current->common.scope != parent) {
217:                 intern->flags |= SPL_FIXEDARRAY_OVERLOADED_CURRENT;
218:             }
219:             if (class_type->iterator_funcs.if_next->common.scope != parent) {
220:                 intern->flags |= SPL_FIXEDARRAY_OVERLOADED_NEXT;
221:             }
222:             intern->fptr_offset_get = zend_hash_str_find_ptr(class_type->function_table, "offsetget", sizeof("offsetget") - 1);
223:             if (intern->fptr_offset_get->common.scope == parent) {
224:                 intern->fptr_offset_get = NULL;
225:             }
226:             intern->fptr_offset_set = zend_hash_str_find_ptr(class_type->function_table, "offsetset", sizeof("offsetset") - 1);
227:             if (intern->fptr_offset_set->common.scope == parent) {
228:                 intern->fptr_offset_set = NULL;
229:             }
230:             intern->fptr_offset_has = zend_hash_str_find_ptr(class_type->function_table, "offsetexists", sizeof("offsetexists") - 1);
231:             if (intern->fptr_offset_has->common.scope == parent) {
232:                 intern->fptr_offset_has = NULL;
233:             }
234:             intern->fptr_offset_del = zend_hash_str_find_ptr(class_type->function_table, "offsetunset", sizeof("offsetunset") - 1);
235:             if (intern->fptr_offset_del->common.scope == parent) {
236:                 intern->fptr_offset_del = NULL;
237:             }
238:             intern->fptr_count = zend_hash_str_find_ptr(class_type->function_table, "count", sizeof("count") - 1);
239:             if (intern->fptr_count->common.scope == parent) {
240:                 intern->fptr_count = NULL;
241:             }
242:         }
243:         return intern->std;
244:     } /* }}} */
245:
246:     static zend_object *spl_fixedarray_new(zend_class_entry *class_type) /* {{{ */ {
247:         return spl_fixedarray_object_new_ce(class_type, NULL, 0);
248:     } /* }}} */
249:
250:     static zend_object *spl_fixedarray_object_clone(zval *object) /* {{{ */ {
251:         zend_object *old_object;
252:         zend_object *new_object;
253:         old_object = Z_OBJ_P(object);
254:         new_object = spl_fixedarray_object_new_ce(old_object->ce, object, 1);
255:         zend_objects_clone_members(new_object, old_object);
256:         return new_object;
257:     } /* }}} */
258:
259:     static inline zval *spl_fixedarray_object_read_dimension_helper(spl_fixedarray_object *intern, zval *offset) /* {{{ */ {
260:         zend_long index;
261:         /* we have to return NULL on error here to avoid memleak because of
262:          * Z_ADDREF_UNINITIALIZED_ZVAL_PTR */
263:         if (isset) {
264:             zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
265:             return NULL;
266:         }
267:         if (IS_LONG_P(offset) && !IS_LONG) {
268:             index = spl_offset_convert_to_long(offset);
269:         } else {
270:             index = Z_LVAL_P(offset);
271:         }
272:         if (index < 0 || (index > intern->array.size)) {
273:             zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
274:             return NULL;
275:         }
276:         if (IS_LONG_P(offset) && !IS_LONG) {
277:             if (!isset) {
278:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
279:                 return NULL;
280:             }
281:             if (isset) {
282:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
283:                 return NULL;
284:             }
285:             if (isset) {
286:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
287:                 return NULL;
288:             }
289:             if (isset) {
290:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
291:                 return NULL;
292:             }
293:             if (isset) {
294:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
295:                 return NULL;
296:             }
297:             if (isset) {
298:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
299:                 return NULL;
300:             }
301:             if (isset) {
302:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
303:                 return NULL;
304:             }
305:             if (isset) {
306:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
307:                 return NULL;
308:             }
309:             if (isset) {
310:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
311:                 return NULL;
312:             }
313:             if (isset) {
314:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
315:                 return NULL;
316:             }
317:             if (isset) {
318:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
319:                 return NULL;
320:             }
321:             if (isset) {
322:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
323:                 return NULL;
324:             }
325:             if (isset) {
326:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
327:                 return NULL;
328:             }
329:             if (isset) {
330:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
331:                 return NULL;
332:             }
333:             if (isset) {
334:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
335:                 return NULL;
336:             }
337:             if (isset) {
338:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
339:                 return NULL;
340:             }
341:             if (isset) {
342:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
343:                 return NULL;
344:             }
345:             if (isset) {
346:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
347:                 return NULL;
348:             }
349:             if (isset) {
350:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
351:                 return NULL;
352:             }
353:             if (isset) {
354:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
355:                 return NULL;
356:             }
357:             if (isset) {
358:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
359:                 return NULL;
360:             }
361:             if (isset) {
362:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
363:                 return NULL;
364:             }
365:             if (isset) {
366:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
367:                 return NULL;
368:             }
369:             if (isset) {
370:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
371:                 return NULL;
372:             }
373:             if (isset) {
374:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
375:                 return NULL;
376:             }
377:             if (isset) {
378:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
379:                 return NULL;
380:             }
381:             if (isset) {
382:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
383:                 return NULL;
384:             }
385:             if (isset) {
386:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
387:                 return NULL;
388:             }
389:             if (isset) {
390:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
391:                 return NULL;
392:             }
393:             if (isset) {
394:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
395:                 return NULL;
396:             }
397:             if (isset) {
398:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
399:                 return NULL;
400:             }
401:             if (isset) {
402:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
403:                 return NULL;
404:             }
405:             if (isset) {
406:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
407:                 return NULL;
408:             }
409:             if (isset) {
410:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
411:                 return NULL;
412:             }
413:             if (isset) {
414:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
415:                 return NULL;
416:             }
417:             if (isset) {
418:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
419:                 return NULL;
420:             }
421:             if (isset) {
422:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
423:                 return NULL;
424:             }
425:             if (isset) {
426:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
427:                 return NULL;
428:             }
429:             if (isset) {
430:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
431:                 return NULL;
432:             }
433:             if (isset) {
434:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
435:                 return NULL;
436:             }
437:             if (isset) {
438:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
439:                 return NULL;
440:             }
441:             if (isset) {
442:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
443:                 return NULL;
444:             }
445:             if (isset) {
446:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
447:                 return NULL;
448:             }
449:             if (isset) {
450:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
451:                 return NULL;
452:             }
453:             if (isset) {
454:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
455:                 return NULL;
456:             }
457:             if (isset) {
458:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
459:                 return NULL;
460:             }
461:             if (isset) {
462:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
463:                 return NULL;
464:             }
465:             if (isset) {
466:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
467:                 return NULL;
468:             }
469:             if (isset) {
470:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
471:                 return NULL;
472:             }
473:             if (isset) {
474:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
475:                 return NULL;
476:             }
477:             if (isset) {
478:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
479:                 return NULL;
480:             }
481:             if (isset) {
482:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
483:                 return NULL;
484:             }
485:             if (isset) {
486:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
487:                 return NULL;
488:             }
489:             if (isset) {
490:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
491:                 return NULL;
492:             }
493:             if (isset) {
494:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
495:                 return NULL;
496:             }
497:             if (isset) {
498:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
499:                 return NULL;
500:             }
501:             if (isset) {
502:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
503:                 return NULL;
504:             }
505:             if (isset) {
506:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
507:                 return NULL;
508:             }
509:             if (isset) {
510:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
511:                 return NULL;
512:             }
513:             if (isset) {
514:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
515:                 return NULL;
516:             }
517:             if (isset) {
518:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
519:                 return NULL;
520:             }
521:             if (isset) {
522:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
523:                 return NULL;
524:             }
525:             if (isset) {
526:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
527:                 return NULL;
528:             }
529:             if (isset) {
530:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
531:                 return NULL;
532:             }
533:             if (isset) {
534:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
535:                 return NULL;
536:             }
537:             if (isset) {
538:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
539:                 return NULL;
540:             }
541:             if (isset) {
542:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
543:                 return NULL;
544:             }
545:             if (isset) {
546:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
547:                 return NULL;
548:             }
549:             if (isset) {
550:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
551:                 return NULL;
552:             }
553:             if (isset) {
554:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
555:                 return NULL;
556:             }
557:             if (isset) {
558:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
559:                 return NULL;
560:             }
561:             if (isset) {
562:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
563:                 return NULL;
564:             }
565:             if (isset) {
566:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
567:                 return NULL;
568:             }
569:             if (isset) {
570:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
571:                 return NULL;
572:             }
573:             if (isset) {
574:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
575:                 return NULL;
576:             }
577:             if (isset) {
578:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
579:                 return NULL;
580:             }
581:             if (isset) {
582:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
583:                 return NULL;
584:             }
585:             if (isset) {
586:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
587:                 return NULL;
588:             }
589:             if (isset) {
590:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
591:                 return NULL;
592:             }
593:             if (isset) {
594:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
595:                 return NULL;
596:             }
597:             if (isset) {
598:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
599:                 return NULL;
600:             }
601:             if (isset) {
602:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
603:                 return NULL;
604:             }
605:             if (isset) {
606:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
607:                 return NULL;
608:             }
609:             if (isset) {
610:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
611:                 return NULL;
612:             }
613:             if (isset) {
614:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
615:                 return NULL;
616:             }
617:             if (isset) {
618:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
619:                 return NULL;
620:             }
621:             if (isset) {
622:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
623:                 return NULL;
624:             }
625:             if (isset) {
626:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
627:                 return NULL;
628:             }
629:             if (isset) {
630:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
631:                 return NULL;
632:             }
633:             if (isset) {
634:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
635:                 return NULL;
636:             }
637:             if (isset) {
638:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
639:                 return NULL;
640:             }
641:             if (isset) {
642:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
643:                 return NULL;
644:             }
645:             if (isset) {
646:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
647:                 return NULL;
648:             }
649:             if (isset) {
650:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
651:                 return NULL;
652:             }
653:             if (isset) {
654:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
655:                 return NULL;
656:             }
657:             if (isset) {
658:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
659:                 return NULL;
660:             }
661:             if (isset) {
662:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
663:                 return NULL;
664:             }
665:             if (isset) {
666:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
667:                 return NULL;
668:             }
669:             if (isset) {
670:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
671:                 return NULL;
672:             }
673:             if (isset) {
674:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
675:                 return NULL;
676:             }
677:             if (isset) {
678:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
679:                 return NULL;
680:             }
681:             if (isset) {
682:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
683:                 return NULL;
684:             }
685:             if (isset) {
686:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
687:                 return NULL;
688:             }
689:             if (isset) {
690:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
691:                 return NULL;
692:             }
693:             if (isset) {
694:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
695:                 return NULL;
696:             }
697:             if (isset) {
698:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
699:                 return NULL;
700:             }
701:             if (isset) {
702:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
703:                 return NULL;
704:             }
705:             if (isset) {
706:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
707:                 return NULL;
708:             }
709:             if (isset) {
710:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
711:                 return NULL;
712:             }
713:             if (isset) {
714:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
715:                 return NULL;
716:             }
717:             if (isset) {
718:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
719:                 return NULL;
720:             }
721:             if (isset) {
722:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
723:                 return NULL;
724:             }
725:             if (isset) {
726:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
727:                 return NULL;
728:             }
729:             if (isset) {
730:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
731:                 return NULL;
732:             }
733:             if (isset) {
734:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
735:                 return NULL;
736:             }
737:             if (isset) {
738:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
739:                 return NULL;
740:             }
741:             if (isset) {
742:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
743:                 return NULL;
744:             }
745:             if (isset) {
746:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
747:                 return NULL;
748:             }
749:             if (isset) {
750:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
751:                 return NULL;
752:             }
753:             if (isset) {
754:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
755:                 return NULL;
756:             }
757:             if (isset) {
758:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
759:                 return NULL;
760:             }
761:             if (isset) {
762:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
763:                 return NULL;
764:             }
765:             if (isset) {
766:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
767:                 return NULL;
768:             }
769:             if (isset) {
770:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
771:                 return NULL;
772:             }
773:             if (isset) {
774:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
775:                 return NULL;
776:             }
777:             if (isset) {
778:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
779:                 return NULL;
780:             }
781:             if (isset) {
782:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
783:                 return NULL;
784:             }
785:             if (isset) {
786:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
787:                 return NULL;
788:             }
789:             if (isset) {
790:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
791:                 return NULL;
792:             }
793:             if (isset) {
794:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
795:                 return NULL;
796:             }
797:             if (isset) {
798:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
799:                 return NULL;
800:             }
801:             if (isset) {
802:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
803:                 return NULL;
804:             }
805:             if (isset) {
806:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
807:                 return NULL;
808:             }
809:             if (isset) {
810:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
811:                 return NULL;
812:             }
813:             if (isset) {
814:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
815:                 return NULL;
816:             }
817:             if (isset) {
818:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
819:                 return NULL;
820:             }
821:             if (isset) {
822:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
823:                 return NULL;
824:             }
825:             if (isset) {
826:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
827:                 return NULL;
828:             }
829:             if (isset) {
830:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
831:                 return NULL;
832:             }
833:             if (isset) {
834:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
835:                 return NULL;
836:             }
837:             if (isset) {
838:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
839:                 return NULL;
840:             }
841:             if (isset) {
842:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
843:                 return NULL;
844:             }
845:             if (isset) {
846:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
847:                 return NULL;
848:             }
849:             if (isset) {
850:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
851:                 return NULL;
852:             }
853:             if (isset) {
854:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
855:                 return NULL;
856:             }
857:             if (isset) {
858:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
859:                 return NULL;
860:             }
861:             if (isset) {
862:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
863:                 return NULL;
864:             }
865:             if (isset) {
866:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
867:                 return NULL;
868:             }
869:             if (isset) {
870:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
871:                 return NULL;
872:             }
873:             if (isset) {
874:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
875:                 return NULL;
876:             }
877:             if (isset) {
878:                 zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
879:                 return NULL;
880:            
```

```

377:     return spl_fixedarray_object_read_dimension_helper(intern, offset);
378: }
379: /* }}} */
380:
381:
382: static inline void spl_fixedarray_object_write_dimension_helper(spl_fixedarray_object *intern, zval *offset, zval *value) /* {{{ */
383: {
384:     zend_long index;
385:
386:     if (!offset) {
387:         /* '[array[] = value]' syntax is not supported */
388:         zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
389:         return;
390:     }
391:
392:     if (Z_TYPE_P(offset) != IS_LONG) {
393:         index = spl_offset_convert_to_long(offset);
394:     } else {
395:         index = Z_LVAL_P(offset);
396:     }
397:
398:     if (index < 0 || index >= intern->array.size) {
399:         zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
400:         return;
401:     } else {
402:         if (!IS_UNDEF(intern->array.elements[index])) {
403:             zval_ptr_dtor(&(intern->array.elements[index]));
404:         }
405:         ZVAL_DEREF(value);
406:         ZVAL_COPY(&(intern->array.elements[index]), value);
407:     }
408: }
409: /* }}} */
410:
411: static void spl_fixedarray_object_write_dimension(zval *object, zval *offset, zval *value) /* {{{ */
412: {
413:     spl_fixedarray_object *intern;
414:     zval tmp;
415:
416:     intern = Z_SPLFIXEDARRAY_P(object);
417:
418:     if (intern->fptr_offset_set) {
419:         if (!offset) {
420:             ZVAL_NULL(&tmp);
421:             offset = &tmp;
422:         } else {
423:             SEPARATE_ARG_IF_REF(offset);
424:         }
425:         SEPARATE_ARG_IF_REF(value);
426:         zend_call_method_with_2_params(object, intern->std.ce, intern->fptr_offset_set, "offsetSet", NULL, offset, value);
427:         zval_ptr_dtor(value);
428:         zval_ptr_dtor(offset);
429:         return;
430:     }
431:
432:     spl_fixedarray_object_write_dimension_helper(intern, offset, value);
433: }
434: /* }}} */
435:
436: static inline void spl_fixedarray_object_unset_dimension_helper(spl_fixedarray_object *intern, zval *offset) /* {{{ */
437: {
438:     zend_long index;
439:
440:     if (Z_TYPE_P(offset) != IS_LONG) {
441:         index = spl_offset_convert_to_long(offset);
442:     } else {
443:         index = Z_LVAL_P(offset);
444:     }
445:
446:     if (index < 0 || index >= intern->array.size) {
447:         zend_throw_exception(spl_ce_RuntimeException, "Index invalid or out of range", 0);
448:         return;
449:     } else {
450:         zval_ptr_dtor(&(intern->array.elements[index]));
451:         ZVAL_UNDEF(&(intern->array.elements[index]));
452:     }
453: }
454: /* }}} */
455:
456: static void spl_fixedarray_object_unset_dimension(zval *object, zval *offset) /* {{{ */
457: {
458:     spl_fixedarray_object *intern;
459:
460:     intern = Z_SPLFIXEDARRAY_P(object);
461:
462:     if (intern->fptr_offset_del) {
463:         SEPARATE_ARG_IF_REF(offset);
464:         zend_call_method_with_2_params(object, intern->std.ce, intern->fptr_offset_del, "offsetUnset", NULL, offset);
465:         zval_ptr_dtor(offset);
466:         return;
467:     }
468:
469:     spl_fixedarray_object_unset_dimension_helper(intern, offset);
470: }
471: }
472: /* }}} */
473:
474: static inline int spl_fixedarray_object_has_dimension_helper(spl_fixedarray_object *intern, zval *offset, int check_empty) /* {{{ */
475: {
476:     zend_long index;
477:     int retval;
478:
479:     if (Z_TYPE_P(offset) != IS_LONG) {
480:         index = spl_offset_convert_to_long(offset);
481:     } else {
482:         index = Z_LVAL_P(offset);
483:     }
484:
485:     if (index < 0 || index >= intern->array.size) {
486:         retval = 0;
487:     } else {
488:         if (!IS_UNDEF(intern->array.elements[index])) {
489:             retval = 1;
490:         } else if (check_empty) {
491:             if (zend_is_true(&(intern->array.elements[index]))) {
492:                 retval = 1;
493:             } else {
494:                 retval = 0;
495:             }
496:         } else /* != NULL and !check_empty */
497:             retval = 1;
498:     }
499: }
500:
501: return retval;
502: }
503: /* }}} */
504:
505: static int spl_fixedarray_object_has_dimension(zval *object, zval *offset, int check_empty) /* {{{ */
506: {
507:     spl_fixedarray_object *intern;
508:
509:     intern = Z_SPLFIXEDARRAY_P(object);
510:
511:     if (intern->fptr_offset_has) {
512:         zval rv;
513:         SEPARATE_ARG_IF_REF(offset);
514:         zend_call_method_with_2_params(object, intern->std.ce, intern->fptr_offset_has, "offsetExists", &rv, offset);
515:         zval_ptr_dtor(&rv);
516:         if (!IS_UNDEF(rv)) {
517:             zend_bool result = zend_is_true(rv);
518:             zval_ptr_dtor(&rv);
519:             return result;
520:         }
521:         return 0;
522:     }
523:
524:     return spl_fixedarray_object_has_dimension_helper(intern, offset, check_empty);
525: }
526: /* }}} */
527:
528: static int spl_fixedarray_object_count_elements(zval *object, zend_long *count) /* {{{ */
529: {
530:     spl_fixedarray_object *intern;
531:
532:     intern = Z_SPLFIXEDARRAY_P(object);
533:     if (intern->fptr_count) {
534:         zval rv;
535:         zend_call_method_with_0_params(object, intern->std.ce, intern->fptr_count, "count", &rv);
536:         if (!IS_UNDEF(rv)) {
537:             *count = zval_get_long(rv);
538:             zval_ptr_dtor(&rv);
539:         } else {
540:             *count = 0;
541:         }
542:     } else {
543:         *count = intern->array.size;
544:     }
545:     return SUCCESS;
546: }
547: /* }}} */
548:
549: /* {{{ proto void SplFixedArray::__construct([int size])
550: */
551: SPL_METHOD(SplFixedArray, __construct)
552: {
553:     zval *object = getThis();
554:     spl_fixedarray_object *intern;
555:     zend_long size = 0;
556:
557:     if (zend_parse_parameters_throw(ZEND_NUM_ARGS(), "|i", &size) == FAILURE) {
558:         return;
559:     }
560:
561:     if (size < 0) {
562:         zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "array size cannot be less than zero");
563:         return;
564:     }

```

```

565:     intern = Z_SPLFIXEDARRAY_P(object);
566:
567:     if (intern->array.size > 0) {
568:         /* called __construct() twice, bail out */
569:         return;
570:     }
571: }
572:
573: spl_fixedarray_init(&(intern->array), size);
574: }
575: /* }}} */
576:
577: /* {{{ proto void SplFixedArray::__wakeup()
578: */
579: SPL_METHOD(SplFixedArray, __wakeup)
580: {
581:     spl_fixedarray_object *intern = Z_SPLFIXEDARRAY_P(getThis());
582:     HashTable *intern_ht = zend_std_get_properties(getThis());
583:     zval *data;
584:
585:     if (zend_parse_parameters_none() == FAILURE) {
586:         return;
587:     }
588:
589:     if (intern->array.size == 0) {
590:         int index = 0;
591:         int size = zend_hash_num_elements(intern_ht);
592:
593:         spl_fixedarray_init(&(intern->array), size);
594:
595:         ZEND_HASH_FOREACH_VAL(intern_ht, data) {
596:             ZVAL_COPY(&(intern->array.elements[index]), data);
597:             index++;
598:         } ZEND_HASH_FOREACH_END();
599:
600:         /* Remove the unserialized properties, since we now have the elements
601          * within the spl_fixedarray_object structure. */
602:         zend_hash_clean(intern_ht);
603:     }
604: }
605: /* }}} */
606:
607: /* {{{ proto int SplFixedArray::count(void)
608: */
609: SPL_METHOD(SplFixedArray, count)
610: {
611:     zval *object = getThis();
612:     spl_fixedarray_object *intern;
613:
614:     if (zend_parse_parameters_none() == FAILURE) {
615:         return;
616:     }
617:
618:     intern = Z_SPLFIXEDARRAY_P(object);
619:     return LONG(intern->array.size);
620: }
621: /* }}} */
622:
623: /* {{{ proto object SplFixedArray::toArray()
624: */
625: SPL_METHOD(SplFixedArray, toArray)
626: {
627:     spl_fixedarray_object *intern;
628:
629:     if (zend_parse_parameters_none() == FAILURE) {
630:         return;
631:     }
632:
633:     intern = Z_SPLFIXEDARRAY_P(getThis());
634:
635:     if (intern->array.size > 0) {
636:         int i = 0;
637:
638:         array_init(&(return_value));
639:         for (; i < intern->array.size; i++) {
640:             if (!IS_UNDEF(intern->array.elements[i])) {
641:                 zend_hash_index_update(&(Z_ARRVAL_P(return_value)), i, &(intern->array.elements[i]));
642:                 Z_TRY_ADDREF(intern->array.elements[i]);
643:             } else {
644:                 zend_hash_index_update(&(Z_ARRVAL_P(return_value)), i, &(Z_UNINITIALIZED_VAL));
645:             }
646:         }
647:     } else {
648:         ZVAL_EMPTY_ARRAY(&(return_value));
649:     }
650: }
651: /* }}} */
652:
653: /* {{{ proto object SplFixedArray::fromArray(array data, bool save_indexes)
654: */
655: SPL_METHOD(SplFixedArray, fromArray)
656: {
657:     zval *data;
658:     spl_fixedarray_array;
659:     spl_fixedarray_object *intern;
660:     int num;
661:     zend_bool save_indexes = 1;
662:
663:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "ab", &data, &save_indexes) == FAILURE) {
664:         return;
665:     }
666:
667:     num = zend_hash_num_elements(&(Z_ARRVAL_P(data)));
668:
669:     if (num > 0 & &save_indexes) {
670:         zval *element;
671:         zend_string *str_index;
672:         zend_long num_index, max_index = 0;
673:         zend_long tmp;
674:
675:         ZEND_HASH_FOREACH_KEY_VAL(&(Z_ARRVAL_P(data)), num_index, str_index) {
676:             if (str_index != NULL) { (zend_long)num_index < 0) {
677:                 zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "array must contain only positive integer keys");
678:                 return;
679:             }
680:
681:             if (num_index > max_index) {
682:                 max_index = num_index;
683:             }
684:         } ZEND_HASH_FOREACH_END();
685:
686:         tmp = max_index + 1;
687:         if (tmp <= 0) {
688:             zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "integer overflow detected");
689:             return;
690:         }
691:         spl_fixedarray_init(&(array), tmp);
692:
693:         ZEND_HASH_FOREACH_KEY_VAL(&(Z_ARRVAL_P(data)), num_index, str_index) {
694:             ZVAL_DEREF(element);
695:             ZVAL_COPY(&(array.elements[num_index]), element);
696:         } ZEND_HASH_FOREACH_END();
697:
698:     } else if (num > 0 & !&save_indexes) {
699:         zval *element;
700:         zend_long i = 0;
701:
702:         spl_fixedarray_init(&(array), num);
703:
704:         ZEND_HASH_FOREACH_VAL(&(Z_ARRVAL_P(data)), element) {
705:             ZVAL_DEREF(element);
706:             ZVAL_COPY(&(array.elements[i]), element);
707:             i++;
708:         } ZEND_HASH_FOREACH_END();
709:     } else {
710:         spl_fixedarray_init(&(array), 0);
711:     }
712:
713:     object_init_ex(&(return_value), spl_ce_SplFixedArray);
714:
715:     intern = Z_SPLFIXEDARRAY_P(&(return_value));
716:     intern->array = array;
717: }
718: /* }}} */
719:
720: /* {{{ proto int SplFixedArray::getSize(void)
721: */
722: SPL_METHOD(SplFixedArray, getSize)
723: {
724:     zval *object = getThis();
725:     spl_fixedarray_object *intern;
726:
727:     if (zend_parse_parameters_none() == FAILURE) {
728:         return;
729:     }
730:
731:     intern = Z_SPLFIXEDARRAY_P(object);
732:     return LONG(intern->array.size);
733: }
734: /* }}} */
735:
736: /* {{{ proto bool SplFixedArray::setSize(int size)
737: */
738: SPL_METHOD(SplFixedArray, setSize)
739: {
740:     zval *object = getThis();
741:     spl_fixedarray_object *intern;
742:     zend_long size;
743:
744:     if (zend_parse_parameters(ZEND_NUM_ARGS(), "i", &size) == FAILURE) {
745:         return;
746:     }
747:
748:     if (size < 0) {
749:         zend_throw_exception_ex(spl_ce_InvalidArgumentException, 0, "array size cannot be less than zero");
750:         return;
751:     }
752:

```

```

753: intern = _splfixedarray_p(object);
754:
755: spl_fixedarray_resize(intern->array, size);
756: RETURN_TRUE;
757: }
758: /* }}} */
759:
760: /* {{{ proto bool SplFixedArray::offsetExists(mixed $index)
761: Returns whether the requested $index exists. */
762: SPL_METHOD(SplFixedArray, offsetExists)
763: {
764:     zval *zindex;
765:     spl_fixedarray_object *intern;
766:
767:     IF_ZEND_PARSE_PARAMETERS(ZEND_NUM_ARGS(), "a", &zindex) == FAILURE {
768:         return;
769:     }
770:
771:     intern = _splfixedarray_p(getThis());
772:
773:     RETURN_BOOL(spl_fixedarray_object_has_dimension_helper(intern, zindex, 0));
774: } /* }}} */
775:
776: /* {{{ proto mixed SplFixedArray::offsetGet(mixed $index)
777: Returns the value at the specified $index. */
778: SPL_METHOD(SplFixedArray, offsetGet)
779: {
780:     zval *zindex, *value;
781:     spl_fixedarray_object *intern;
782:
783:     IF_ZEND_PARSE_PARAMETERS(ZEND_NUM_ARGS(), "a", &zindex) == FAILURE {
784:         return;
785:     }
786:
787:     intern = _splfixedarray_p(getThis());
788:     value = spl_fixedarray_object_read_dimension_helper(intern, zindex);
789:
790:     IF (value) {
791:         ZVAL_DEREF(value);
792:         ZVAL_COPY(return_value, value);
793:     } else {
794:         RETURN_NULL();
795:     }
796: } /* }}} */
797:
798: /* {{{ proto void SplFixedArray::offsetSet(mixed $index, mixed $newval)
799: Sets the value at the specified $index to $newval. */
800: SPL_METHOD(SplFixedArray, offsetSet)
801: {
802:     zval *zindex, *value;
803:     spl_fixedarray_object *intern;
804:
805:     IF_ZEND_PARSE_PARAMETERS(ZEND_NUM_ARGS(), "aa", &zindex, &value) == FAILURE {
806:         return;
807:     }
808:
809:     intern = _splfixedarray_p(getThis());
810:     spl_fixedarray_object_write_dimension_helper(intern, zindex, value);
811:
812: } /* }}} */
813:
814: /* {{{ proto void SplFixedArray::offsetUnset(mixed $index)
815: Unsets the value at the specified $index. */
816: SPL_METHOD(SplFixedArray, offsetUnset)
817: {
818:     zval *zindex;
819:     spl_fixedarray_object *intern;
820:
821:     IF_ZEND_PARSE_PARAMETERS(ZEND_NUM_ARGS(), "a", &zindex) == FAILURE {
822:         return;
823:     }
824:
825:     intern = _splfixedarray_p(getThis());
826:     spl_fixedarray_object_unset_dimension_helper(intern, zindex);
827:
828: } /* }}} */
829:
830: static void spl_fixedarray_it_dtor(zend_object_iterator *iter) /* {{{ */
831: {
832:     spl_fixedarray_it *iterator = (spl_fixedarray_it *)iter;
833:
834:     zend_user_it_invalidate_current(iter);
835:     zval_ptr_dtor(&iterator->intern.it.data);
836: } /* }}} */
837:
838: static void spl_fixedarray_it_rewind(zend_object_iterator *iter) /* {{{ */
839: {
840:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
841:
842:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_REWIND) {
843:         zend_user_it_rewind(iter);
844:     } else {
845:         object->current = 0;
846:     }
847: } /* }}} */
848:
849: static int spl_fixedarray_it_valid(zend_object_iterator *iter) /* {{{ */
850: {
851:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
852:
853:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_VALID) {
854:         return zend_user_it_validate(iter);
855:     }
856:
857:     IF (object->current >= 0 && object->current < object->array.size) {
858:         return SUCCESS;
859:     }
860:     return FAILURE;
861: } /* }}} */
862:
863: static zval *spl_fixedarray_it_get_current_data(zend_object_iterator *iter) /* {{{ */
864: {
865:     zval *zindex;
866:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
867:
868:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_CURRENT) {
869:         return zend_user_it_get_current_data(iter);
870:     } else {
871:         zval *data;
872:
873:         ZVAL_LONG(&zindex, object->current);
874:
875:         data = spl_fixedarray_object_read_dimension_helper(object, &zindex);
876:         zval_ptr_dtor(&zindex);
877:
878:         IF (data == NULL) {
879:             data = &EG(uninitialized_zval);
880:         }
881:         return data;
882:     }
883: } /* }}} */
884:
885: static void spl_fixedarray_it_get_current_key(zend_object_iterator *iter, zval *key) /* {{{ */
886: {
887:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
888:
889:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_KEY) {
890:         zend_user_it_get_current_key(iter, key);
891:     } else {
892:         ZVAL_LONG(key, object->current);
893:     }
894: } /* }}} */
895:
896: static void spl_fixedarray_it_move_forward(zend_object_iterator *iter) /* {{{ */
897: {
898:     spl_fixedarray_object *object = _splfixedarray_p(iterator->data);
899:
900:     IF (object->flags & SPL_FIXEDARRAY_OVERLOADED_NEXT) {
901:         zend_user_it_move_forward(iter);
902:     } else {
903:         zend_user_it_invalidate_current(iter);
904:         object->current++;
905:     }
906: } /* }}} */
907:
908: /* {{{ proto int SplFixedArray::key()
909: Return current array key */
910: SPL_METHOD(SplFixedArray, key)
911: {
912:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
913:
914:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
915:         return;
916:     }
917:
918:     RETURN_LONG(intern->current);
919: } /* }}} */
920:
921: /* {{{ proto void SplFixedArray::next()
922: Move to next entry */
923: SPL_METHOD(SplFixedArray, next)
924: {
925:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
926:
927:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
928:         return;
929:     }
930:
931:     RETURN_LONG(intern->current++);
932: }

```

```

941: /* }}} */
942:
943: /* {{{ proto bool SplFixedArray::valid()
944: Check whether the datastructure contains more entries */
945: SPL_METHOD(SplFixedArray, valid)
946: {
947:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
948:
949:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
950:         return;
951:     }
952:
953:     RETURN_BOOL(intern->current >= 0 && intern->current < intern->array.size);
954: } /* }}} */
955:
956: /* {{{ proto void SplFixedArray::rewind()
957: Rewind the datastructure back to the start */
958: SPL_METHOD(SplFixedArray, rewind)
959: {
960:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
961:
962:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
963:         return;
964:     }
965:
966:     intern->current = 0;
967: } /* }}} */
968:
969: /* {{{ proto mixed SplFixedArray::current()
970: Return current datastructure entry */
971: SPL_METHOD(SplFixedArray, current)
972: {
973:     zval *zindex, *value;
974:     spl_fixedarray_object *intern = _splfixedarray_p(getThis());
975:
976:     IF_ZEND_PARSE_PARAMETERS_NONE() == FAILURE {
977:         return;
978:     }
979:
980:     ZVAL_LONG(&zindex, intern->current);
981:
982:     value = spl_fixedarray_object_read_dimension_helper(intern, &zindex);
983:
984:     zval_ptr_dtor(&zindex);
985:
986:     IF (value) {
987:         ZVAL_DEREF(value);
988:         ZVAL_COPY(return_value, value);
989:     } else {
990:         RETURN_NULL();
991:     }
992: } /* }}} */
993:
994: /* Iterator handler table */
995: static const zend_object_iterator_funcs spl_fixedarray_it_funcs = {
996:     spl_fixedarray_it_dtor,
997:     spl_fixedarray_it_valid,
998:     spl_fixedarray_it_get_current_data,
999:     spl_fixedarray_it_get_current_key,
1000:     spl_fixedarray_it_move_forward,
1001:     spl_fixedarray_it_rewind,
1002:     NULL
1003: };
1004:
1005: zend_object_iterator *spl_fixedarray_get_iterator(zend_class_entry *ce, zval *object, int by_ref) /* {{{ */
1006: {
1007:     spl_fixedarray_it *iterator;
1008:
1009:     IF (by_ref) {
1010:         zend_throw_exception(spl_ce_RuntimeException, "An iterator cannot be used with foreach by reference", 0);
1011:         return NULL;
1012:     }
1013:
1014:     iterator = emalloc(sizeof(spl_fixedarray_it));
1015:
1016:     zend_iterator_init(&zend_object_iterator*(iterator));
1017:
1018:     ZVAL_COPY(&iterator->intern.it.data, object);
1019:     iterator->intern.it.funcs = &spl_fixedarray_it_funcs;
1020:     iterator->intern.ce = ce;
1021:     ZVAL_UNDEF(&iterator->intern.value);
1022:
1023:     return iterator->intern.it;
1024: } /* }}} */
1025:
1026: /* {{{ */
1027:
1028: /* {{{ */
1029:
1030: ZEND_ARG_INFO_EX(arginfo_splfixedarray_construct, 0, 0, 0)
1031: ZEND_ARG_INFO(0, size)
1032: ZEND_ARG_INFO(0, offset)
1033:
1034: ZEND_ARG_INFO_EX(arginfo_splfixedarray_offsetGet, 0, 0, 1)
1035: ZEND_ARG_INFO(0, index)
1036: ZEND_ARG_INFO(0, offset)
1037:
1038: ZEND_ARG_INFO_EX(arginfo_splfixedarray_offsetSet, 0, 0, 2)
1039: ZEND_ARG_INFO(0, index)
1040: ZEND_ARG_INFO(0, newval)
1041: ZEND_ARG_INFO(0, offset)
1042:
1043: ZEND_ARG_INFO_EX(arginfo_splfixedarray_setSize, 0)
1044: ZEND_ARG_INFO(0, value)
1045: ZEND_ARG_INFO(0, offset)
1046:
1047: ZEND_ARG_INFO_EX(arginfo_splfixedarray_fromArray, 0, 0, 1)
1048: ZEND_ARG_INFO(0, data)
1049: ZEND_ARG_INFO(0, save_indexes)
1050: ZEND_ARG_INFO(0, offset)
1051:
1052: ZEND_ARG_INFO_EX(arginfo_splfixedarray_void, 0)
1053: ZEND_ARG_INFO(0, offset)
1054:
1055: static const zend_function_entry spl_funcs_splfixedarray[] = { /* {{{ */
1056:     SPL_ME(splfixedarray, __construct, arginfo_splfixedarray_construct, ZEND_ACC_PUBLIC)
1057:     SPL_ME(splfixedarray, __wakeup, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1058:     SPL_ME(splfixedarray, __clone, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1059:     SPL_ME(splfixedarray, toArray, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1060:     SPL_ME(splfixedarray, fromArray, arginfo_splfixedarray_fromArray, ZEND_ACC_PUBLIC | ZEND_ACC_STATIC)
1061:     SPL_ME(splfixedarray, setSize, arginfo_splfixedarray_setSize, ZEND_ACC_PUBLIC)
1062:     SPL_ME(splfixedarray, setSize, arginfo_splfixedarray_setSize, ZEND_ACC_PUBLIC)
1063:     SPL_ME(splfixedarray, offsetExists, arginfo_splfixedarray_offsetGet, ZEND_ACC_PUBLIC)
1064:     SPL_ME(splfixedarray, offsetGet, arginfo_splfixedarray_offsetGet, ZEND_ACC_PUBLIC)
1065:     SPL_ME(splfixedarray, offsetSet, arginfo_splfixedarray_offsetSet, ZEND_ACC_PUBLIC)
1066:     SPL_ME(splfixedarray, offsetUnset, arginfo_splfixedarray_offsetSet, ZEND_ACC_PUBLIC)
1067:     SPL_ME(splfixedarray, rewind, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1068:     SPL_ME(splfixedarray, key, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1069:     SPL_ME(splfixedarray, next, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1070:     SPL_ME(splfixedarray, valid, arginfo_splfixedarray_void, ZEND_ACC_PUBLIC)
1071:     PHP_FE_END
1072: };
1073:
1074: /* {{{ PHP_MINIT_FUNCTION */
1075:
1076: /* {{{ PHP_MINIT_FUNCTION(spl_fixedarray)
1077:
1078: {
1079:     REGISTER_SPL_STM_CLASS_EX(SplFixedArray, spl_fixedarray_new, spl_funcs_splfixedarray);
1080:     memory(spl_handler_splfixedarray, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
1081:
1082:     spl_handler_splfixedarray.offset = &offsetOf(spl_fixedarray_object, scd);
1083:     spl_handler_splfixedarray.clone_obj = &spl_fixedarray_object_clone;
1084:     spl_handler_splfixedarray.read_dimension = &spl_fixedarray_object_read_dimension;
1085:     spl_handler_splfixedarray.write_dimension = &spl_fixedarray_object_write_dimension;
1086:     spl_handler_splfixedarray.unset_dimension = &spl_fixedarray_object_unset_dimension;
1087:     spl_handler_splfixedarray.has_dimension = &spl_fixedarray_object_has_dimension;
1088:     spl_handler_splfixedarray.count_elements = &spl_fixedarray_object_count_elements;
1089:     spl_handler_splfixedarray.get_properties = &spl_fixedarray_object_get_properties;
1090:     spl_handler_splfixedarray.get_gc = &spl_fixedarray_object_get_gc;
1091:     spl_handler_splfixedarray.dtor_obj = &zend_object_dtor_obj;
1092:     spl_handler_splfixedarray.free_obj = &spl_fixedarray_object_free_storage;
1093:
1094:     REGISTER_SPL_IMPLEMENT(SplFixedArray, Iterator);
1095:     REGISTER_SPL_IMPLEMENT(SplFixedArray, ArrayAccess);
1096:     REGISTER_SPL_IMPLEMENT(SplFixedArray, Countable);
1097:
1098:     spl_ce_splfixedarray->get_iterator = &spl_fixedarray_get_iterator;
1099:
1100:     return SUCCESS;
1101: }
1102:
1103: /* }}} */
1104:
1105: /*
1106: * Local variables:
1107: * tab-width: 4
1108: * c-basic-offset: 4
1109: * End
1110: * vim600: noet sw=4 ts=4 fdm=marker
1111: * vim<600: noet sw=4 ts=4
1112: */

```



```
1: /*
2:  *-----
3:  * | PHP Version 7 |
4:  *-----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  *-----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following uri: |
10:  * | http://www.php.net/license/3_01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  *-----
15:  * | Author: Antony Duvall <trond@daylesday.org> |
16:  * | Elienne Kneuss <colder@php.net> |
17:  *-----
18:  */
19:
20: /* $Id$ */
21:
22: #ifndef SPL_FIXEDARRAY_H
23: #define SPL_FIXEDARRAY_H
24:
25: extern PHPAPI zend_class_entry *spl_ce_SplFixedArray;
26:
27: PHP_MINIT_FUNCTION(spl_fixedarray);
28:
29: #endif /* SPL_FIXEDARRAY_H */
30:
31: /*
32:  * Local Variables:
33:  * tab-width: 4
34:  * c-basic-offset: 4
35:  * End:
36:  * vim600: noet sw=4 ts=4 fM=marker
37:  * vim600: noet sw=4 ts=4
38:  */
```

```
1: /*
2:  * -----
3:  * | PHP Version 7 |
4:  * -----
5:  * | Copyright (c) 1997-2018 The PHP Group |
6:  * -----
7:  * | This source file is subject to version 3.01 of the PHP license, |
8:  * | that is bundled with this package in the file LICENSE, and is |
9:  * | available through the world-wide-web at the following url: |
10:  * | http://www.php.net/license/3.01.txt |
11:  * | If you did not receive a copy of the PHP license and are unable to |
12:  * | obtain it through the world-wide-web, please send a note to |
13:  * | license@php.net so we can mail you a copy immediately. |
14:  * -----
15:  * | Authors: Marcus Boerger <helly@php.net> |
16:  * -----
17:  */
18:
19: #ifndef PHP_SPL_H
20: #define PHP_SPL_H
21:
22: #include "php.h"
23: #include "stdarg.h"
24:
25: #define PHP_SPL_VERSION PHP_VERSION
26:
27: #extern zend_module_entry spl_module_entry;
28: #define phpext_spl_ptr spl_module_entry
29:
30: #ifndef PHP_WIN32
31: # if defined SPL_EXPORTS
32: #  define SPL_API __declspec(dllexport)
33: # elif defined(COMPILE_DL_SPL)
34: #  define SPL_API __declspec(dllimport)
35: # else
36: #  define SPL_API /* nothing */
37: # endif
38: #elif defined(__GNUC__) && __GNUC__ >= 4
39: # define SPL_API __attribute__((visibility("default")))
40: #else
41: # define SPL_API
42: #endif
43:
44: #if defined(PHP_WIN32) && !defined(COMPILE_DL_SPL)
45: #undef phpext_spl
46: #define phpext_spl NULL
47: #endif
48:
49: #PHP_MINIT_FUNCTION(spl);
50: #PHP_MSHUTDOWN_FUNCTION(spl);
51: #PHP_RINIT_FUNCTION(spl);
52: #PHP_ASHUTDOWN_FUNCTION(spl);
53: #PHP_MINFO_FUNCTION(spl);
54:
55:
56: #END_BEGIN_MODULE_GLOBALS(spl)
57: zend_string *autoload_extensions;
58: HashTable *autoload_functions;
59: intptr_t hash_mask_handle;
60: intptr_t hash_mask_handlers;
61: int hash_mask_init;
62: int autoload_running;
63: #END_END_MODULE_GLOBALS(spl)
64:
65: #END_EXTERN_MODULE_GLOBALS(spl)
66: #define SPL_G(v) ZEND_MODULE_GLOBALS_ACCESSOR(spl, v)
67:
68: #PHP_FUNCTION(spl_classes);
69: #PHP_FUNCTION(class_parents);
70: #PHP_FUNCTION(class_implements);
71: #PHP_FUNCTION(class_used);
72:
73: #PHPAPI zend_string *php_spl_object_hash(zval *obj);
74:
75: #endif /* PHP_SPL_H */
76:
77: /*
78:  * Local Variables:
79:  * -tab-width: 4
80:  * -tab-width: 4
81:  * End:
82:  * vim: set fdm=marker
83:  * vim: noet sw=4 ts=4
84:  */
```